Masoume M. Raeissi

Modeling Supervisory Control in Multi-Robot Applications

Ph.D. Thesis

January 24, 2018

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
prof. Alessandro Farinelli

Series N°: **???? (ask the PhD coordinator!)**

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

Abstract

Summary. We consider multi-robot applications, where a human operator monitors and supervise the team to pursue complex objectives in complex environments. Robots, specially at field sites, are often subject to unexpected events that can not be managed without the intervention of the operator(s). For example, in an environmental monitoring application, robots might face extreme environmental events (e.g. water currents) or moving obstacles (e.g. animal approaching the robots). In such scenarios, the operator often needs to interrupt the activities of individual team members to deal with particular situations. This work focuses on human-multi-robot-interaction in these casts. A widely used approach to monitor and supervise robotic teams are team plans, which allow an operator to interact via high level objectives and use automation to work out the details.

The first problem we address in this context, is how human interrupts (i.e. change of action due to unexpected events) can be handled within a robotic team. Typically, after such interrupts, the operator would need to restart the team plan to ensure its success. This causes delays and imposes extra load on the operator. We address this problem by presenting an approach to encoding how interrupts can be smoothly handled within a team plan. Building on a team plan formalism that uses Colored Petri Nets, we describe a mechanism that allows a range of interrupts to be handled smoothly, allowing the team to effectively continue with its task after the operator intervention.

We validate the approach with an application of robotic water monitoring. Our experiments show that the use of our interrupt mechanism decreases the time to complete the plan (up to 48% reduction) and decreases the operator load (up to 80% reduction in number of user actions). Moreover, we performed experiments with real robotic platforms to validate the applicability of our mechanism in the actual deployment of robotic watercraft.

The second problem we address is how to handle intervention requests from robots to the operator. In this case, we consider autonomous robotic platforms that are able to identify their situation and ask for the intervention of the operator by sending a request. However, large teams can easily overwhelm the operator with several requests, hence hindering the team performance. As a consequence, team members will have to wait for the operator attention, and the operator becomes a bottleneck for the system. Our contribution in this context is to make the robots learn cooperative strategies to best utilize the operator's time and decrease the idle time of the robotic system. In particular, we consider a queuing model (a.k.a balking queue), where robots decide whether or not to join the queue. Such decisions are computed by considering dynamic features of the system (e.g. the severity of the request, number of requests, etc.).

We examine several decision making solutions for computing these cooperative strategies, where our goal is to find a trade-off between lower idle time by joining the queue and fewer failures due to the risk of not joining the queue.

We validate the proposed approaches in a simulation robotic water monitoring application. The obtained results show the effectiveness of our proposed models in comparison to the queue without balking, when considering team reward and total idle time.

Contents

1

Introduction

Using robotic solutions is becoming increasingly popular in real-world applications. Robots can assist humans in dangerous applications, such as search and rescue [18, 47, 75] or in repetitive tasks, such as environmental monitoring [27, 74]. In such applications, using multi-robot solutions often brings numerous advantages to the system over the single-robot scenarios, including robustness and effectiveness. For example, multiple robots in an environmental monitoring application can cover a given area faster (i.e more time-efficient) or a broken robot which is unable to finish its task, can be replaced by other robots (i.e. robustness). Finally, multiple robots take advantages of distributed sensors and actuators, hence they can perform complex tasks that is impossible or too hard for a single robot to accomplish.

Usually one or a few number of operators are required to interact with the team of robots to achieve flexible and robust behaviors. Including one or a few number of human operators in the team provides many benefits especially for real-world applications with low-cost robotic platforms. In these cases, while the robots are mostly autonomous, they will be monitored and controlled by the operator(s). In such scenarios, the operator intermittently needs to directly control a robot to protect it from a danger it cannot perceive or to change the current objective. For example, in an environmental monitoring application, after some data has been acquired, the operator may find some areas (e.g. an area around a drain pipe in a river) more interesting than others and sends a particular robot to explore those areas.

Our motivation domain for this work is a team of robotic boats collecting information on bodies of water [60]. In such applications, one or a small number of experienced operators, perhaps water scientists, are managing between five and twenty five boats on a body of water. Large manned boats and water phenomena are external dangers to the robots that the human operators might be able to help mitigate. In other cases, operators might have some external knowledge about what is going on in the water that allows them to direct resources in a very specific way to get very specific information.

In this thesis, we address two important issues related to human-multi-robot supervision. The first one is how to handle interrupts (i.e. a danger that the team can not perceive) that operators might need to make so to effectively supervise

the team. The second one is how to manage requests, that robots can submit to the operator, when they need human intervention.

1.1 Team Plan

As mentioned before, interrupts from human operator to the robots are a crucial aspect for our reference scenario.

For example, consider a situation where the team of boats is instructed to acquire measurements in a set of prespecified locations. Each boat is assigned to a subset of such locations and all boats execute their plan in parallel. If one of the boats must be pulled out from the plan (e.g., to recharge the battery), the other boats should continue their task without stopping. In another situation, the operators might want to slightly change the course of actions of the entire team (e.g., reassign tasks to all available boats when one is pulled out) or even drastically change the current plan of all boats to handle a dangerous situation (e.g., a manned boat suddenly enters the area of operation). The key focus of this work is to provide a general mechanism to handle all the above situations without aborting and restarting the current plan.

Besides the hardware and software challenges in designing single robots, new challenges arise when developing multi-robot systems. Task allocations, coordinations, cooperations and communications are some examples of those challenges. As a result, the need for a strong software architecture is essential in designing multi-robot systems.

A common approach for designing multi-robot systems is applying team plan, which provides a formal language to specify the actions for the whole team. The team-level specification must be responsible for team level operations, such as defining the team's behavior, task allocation, monitoring robots' activities, interacting with members and handling unexpected events. While, the low level software (i.e. the code running on each robot) is responsible for single robot behaviors such as localization, motion planning, etc. For example, a team plan for environmental monitoring might tell robots to collect a certain type of information in a certain area, leaving the robots to work out how to collect the information.

The problem of monitoring plan execution in multi robot systems has been studied in the literature. Two successful BDI-based frameworks for plan specification are STEAM [70] and BITE [40], which enable a coherent teamwork structure for multiple agents. However, they do not provide any specific mechanism for interrupting the execution of such plans. There is substantial literature on the topic of using Petri Nets [52] and variants such as Colored Petri Nets [37] as the basis for representing team plans. For example, [80] proposed an approach for plan monitoring called Petri Net Plans (PNPs). One important functionality offered by the formalism of PNP is the possibility to modify the execution of a plan at run-time using interrupts. While PNP framework provides facilities for handling unexpected events, it does not explicitly consider the involvement of human operators and their intervention in case of robot failures or unexpected events.

Typically, when a robot plan is interrupted, any team plan that the robot was participating in will be terminally impacted. In some cases, the rest of the

team can reorganize without the interrupted robot and then reorganize when the interrupt is over, but this depends on the plan, the particular situation, and nature of the interruption. In general, how to respond to an external interruption heavily depends on the specific context of the plan and if the context is not taken into account when dealing with the interruption, overall performance will be poor.

To realize these sophisticated interactions, we adopt an approach for creating team plans with Petri Nets that allow specification of complex, parallel, and hierarchical plans. Depending on the nature and timing of the interaction, relative to the specific context of the plan, the expressive approach allows for a range of possibilities to be encoded, including restarting the plan, directly resuming, or going through some intermediate steps to restart effectively. The key is that the plan designer can work out in advance how to handle interruptions at a particular place in the plan and encode efficient and effective resumptions.

We validate our approach within the application of robotic watercraft. In particular, we consider a situation where several platforms should travel through a set of pre-specified locations, and we identify three specific cases that require the operator to interrupt the plan execution: (i) a boat must be pulled out; (ii) all boats should stop the plan and move to a pre-specified assembly position; (iii) a set of boats must synchronize to traverse a dangerous area one after the other, so that the human operator can closely monitor the behavior of each single boat and tele-operate the platform if necessary.

For each of these incidents, we compared the execution of team plans without specific interrupt handling to plans where interrupt handlers were explicitly encoded by using our framework. We found significant improvement in overall efficiency. More specifically, the experiments show that the use of our interrupt mechanism decreases the time to complete the plan (up to 48% reduction) and decreases the operator load (up to 80% reduction in number of user actions). Moreover, we performed experiments with real robotic platforms to validate the applicability of our mechanism in the actual deployment of robotic watercraft. Such experiments indicate that our mechanism can be effectively used in actual operations. We present the details of our interrupt mechanism and the corresponding results in Chapter 5.

1.2 Self-Reflection

While the interrupt mechanism improves the efficiency of the system, however the monitoring role of the operator can become critical when the team size grows. To decrease the operator's workload and increase the overall team performance, several approaches have considered the concept of self-reflection, where robots initiate an interaction with the operator when needed. For example, the robot can perceive that its battery level is in a critical state, then it can send a request for the operator's intervention. In our scenario, adding self-reflection to robotic boats (e.g. by warning the operator or asking his/her permission) helps the operator to be more focused on his/her controlling role. However, several boats may need the operator's attention and the operator cannot handle all requests simultaneously. Thus, the requests will be queued and addressed sequentially.

A natural way to deal with this situation is to apply queue theory to multi-agent systems [20, 42]. The main focus of such previous work is on investigating different queue disciplines (i.e. the order in which the requests should be processed by the operator). For example, [20, 42] examine and compare FIFO and SJF queuing models where the requests will be queued according to their arrival time and shortest service time respectively, while [58] proposes using a priority queue in which an assistant agent rearranges the requests and offers the highest priority task to the operator. Since the queue-related autonomy of robots was not addressed in those work, the queue size may grow indefinitely as no robot will leave the queue before receiving the operator's attention. Keeping robots idle until the operator is available might impact team performance, since it can significantly delay the operations of robots waiting in the queue.

To deal with this problem, we focus on a specific queuing model with a balking property in which the users/agents (i.e robots requesting attention in our domain) can decide either to join the queue or balk [46]. Such decision is typically based on a threshold value that is computed by assigning a generic reward associated with receiving the service and a cost for waiting in the queue to each agent. However, in this model [46], there is no gain or loss associated with the balking action. The agent is willing to join the queue if it expects that the cost of waiting for service will be no more than the value obtained from the service. When applying this model to a robotic application, there is no clear indication on how such a threshold can be computed. More important, this model does not consider the cost of balking (i.e. the cost of a potential failure that the robot can have by trying to overcome a difficult situation without human intervention).

Moreover, in a multi-robot scenario, each arrival is a request from a robot not a robot itself which means one robot can have various requests types with different severity. Therefore, different rewards and costs for each type of request should be considered. For example, a robot with a high severity request cannot balk the queue only because the queue is too long. In addition, balking has a cost for the team not only for the robot. Because, the robot that balks a request may not be able to accomplish its assigned task(s). In this case, the remaining task(s) of that robot should be reassigned to the other robots which brings extra loads to both the operator and other team members. Finally, the balking or joining decision for each arrival (request) is not a one-step decision making procedure but a sequential decision making process. For example, the decision of joining the queue will affect the future decisions of the other robots, because the queue size will increase and may become greater than the expected threshold queue size of the next robot with a request. While, choosing to balk with some probability (regarding to request type) may result in failure which will affect the performance of other members and the operator as mentioned before.

Within this context, our goal is to make the team of robots learn cooperative balking strategies to make better use of the shared queue. However, due to the unknown dynamic, non-deterministic environment (e.g. uncertainty in the state transition function, unknown operator's availability and skills, etc.) and partial observability, defining appropriate behavior for each robot is not trivial. The balking strategies must tell each robot for each state of the system whether to join the queue or not, while each robot only observes part of the world's state.

Reinforcement Learning [68] is a common solution in robotic systems, in which the robots interact with their environment to automatically determine the ideal behavior within a particular context. More specifically, each robot learns the proper behavior from the consequences of its actions. Considering this, we present three models for computing balking policies, which starts by a simple dynamic threshold computation and will be evolved to a single-robot reinforcement learning approach and multi-robot reinforcement learning approach that computes the cooperative balking strategies for each robot. We compare queuing structures FIFO and SJF (without balking) with our balking models and illustrate the considerable effectiveness of our proposed models with respect to the team reward and total idle time. We present the details of each model and the corresponding results in Chapter 6.

1.3 Thesis Contributions

The main goal of this thesis is to improve human-multi-robot interactions from two perspectives. First, by providing a general and smooth interrupt mechanism we aim at decreasing the operator's workload and decreasing the time to complete the plan. Second, we consider the situations where the robots are able to identify their needs and ask the operator for help. Then, we provide solutions that allow the robots to decide when and which requests must be sent to the operator. Our goal here is to decrease the monitoring workload of the operator while decreasing the idle time of the system (i.e. the time that the robots have to wait for the operator).

In more details, the main contributions of the thesis are the followings:

1. We present an approach to encoding how interrupts can be smoothly handled within a team plan. Building on a team plan formalism that uses Colored Petri Nets, we describe a general mechanism that allows a range of interrupts to be handled smoothly, allowing the team to efficiently continue with its task after the operator intervention. This contribution has been published in [25] and [26].
2. We model the human-multi-robot interactions as a balking queue in which the robots decide to interrupt the human operator (by joining the queue) or not (balking). We investigate different solutions for learning balking strategies where robots decide when to join and when to balk. This contribution has been published in [56].
3. We evaluate the performance of our proposed models for general interrupt mechanism and balking queue strategies in a multi-robot water monitoring application.

1.4 Thesis Structure

The rest of this document is organized as follows:

Chapter 2: This chapter provides the state of the arts for team plans specification. We position our work with respect to the existing literature, considering two main

groups: approaches which are based on Beliefs-Desire-Intention (BDI) [41, 55, 70] and approaches which are based on Petri Net (PN) [37, 52, 80].

Chapter 3: This chapter first provides the state of the arts for robots with self-reflection and autonomy. Then, we describe Markov Decision Process (MDP) [7] and Decentralized Markov Decision Process (Dec-MDP) [10, 30] which are widley used frameworks for decision making under uncertainty. Next, we discuss Multi-Robot Reinforcement Learning and specifically Q-Learning [68, 76]. Finally, we present a brief introduction to the *Balking Queue* [46] model.

Chapter 4: This chapter presents our motivation domain, the robotic boat system. We explain the hardware architecture of the robotic platform, the plan specification language and monitoring framework used in such system. This robotic system, which is part of INTCATCH project [1], aims at demonstrating the use of low-cost robotic boats for water monitoring.

Chapter 5: This chapter details our proposed interrupt mechanisms which are built on team plan specification. We provide a variety of experiments in simulation and real-world, with and without interrupt mechanism, and show how our model can enhance the performance of the system.

Chapter 6: This chapter presents our solutions for mapping the human-multi-robot interactions into a balking queue structure. Then, we present our approach, cooperative decision making in this scenario. We discuss the empirical methodologies and obtained results.

Chapter 7: This chapter concludes the thesis with a brief summary of our research contributions and the possible directions for future work.

[1] `http://intcatch.eu/index.php`

Part I

Background: Interacting with Multi-Robot Systems

2

State of the Art: Approaches for Interaction with Multi-Robot Systems

Robotics technology has matured sufficiently to make the idea of building robot teams for real environments, including disaster response [18, 47, 75], environmental monitoring [27, 74], surveillance [36, 50] and agricultural operations [3], a serious possibility. Although there are a wide range of studies on different aspects of multi-robot systems, such as coordination, task assignments, communications and so on, the focus of our work is on the interaction with such systems to manage and handle unexpected situations. In these environments, team members often access to incomplete and possibly inconsistent views of the world and the state of other robots. Furthermore, particular team members are often subject to unexpected events due to high uncertainties and complex dynamic of such domains. Therefore, it is difficult to anticipate and pre-plan for all possible events. Hence, we are looking at solutions which provide the team with flexible reactions when encounter any problems. An effective way of doing this is via team plans [40, 70, 80] that allow monitoring member's behavior, interacting with them and reorganizing the team when needed. Moreover, in most real domains, human operators will occasionally need to directly control a robot for some purpose, perhaps to protect a robot from a danger it cannot perceive or to achieve some specific objectives that the robot is not capable of understanding. Team plans allow an operator to specify high level directives and allow the team to autonomously determine how to implement such directives. Hence, in this chapter, we review the related work that concentrate on this aspect of multi-robot systems and the use of team plans in multi-robot applications.

2.1 BDI-Based Plan Representation

STEAM (a Shell for TEAMwork) [70] presents a general model of teamwork. The key aspect of STEAM is team operators, which are based on Joint Intention Theory [21]. In STEAM, agents can monitor the team's performance and reorganize the team based on the current situation. STEAM facilitates monitoring of team performance by exploiting explicit representation of team goals and plans. If individuals responsible for particular subtasks fail in fulfilling their responsibilities, or if new tasks are discovered without an appropriate assignment of team mem-

bers to fulfill them, team reorganization can occur. Such reorganization, as well as recovery from failures in general, is also driven by the team's join intentions.

TEAMCORE [55] architecture which extends the STEAM framework focuses on minimizing the complexity of building flexible teams via a domain-independent infrastructure to support team-oriented programming (TOP). Each TEAMCORE agent has a corresponding proxy which serves as a middle layer between the TOP framework and domain-level agents. The proxy captures the capabilities of its agent and handles communication and coordination between other team members, and adds support for heterogeneous and distributed teams.

Machinetta [61] framework makes further improvements over the proxy concepts by providing each agent with a proxy that has teamwork knowledge hence, allowing for larger teams of agents to work together. The specific mechanism to accomplish a particular goal are left to the agent, allowing for plans to be specified at a high level independent of the actual agents which will ultimately fulfill it.

BITE [40,41] provides integrated synchronization and allocation for team, while previous works have addressed one aspect at a time. In more details, BITE separates task behaviors that control a robot's interaction with its task, from interaction behaviors that control a robot's interaction with its teammates. It also specifies a library of social behaviors and offers different synchronization protocols that can be used interchangeably and mixed as needed. Inspired by STEAM, BITE also maintains a organizational hierarchy and goal behavior graph. One key addition in the BITE architecture is the introduction of a library of hierarchically linked social interaction behaviors implementing interaction protocols for synchronization and task allocation. The goal behavior graph allows specifying which synchronization or task allocation algorithm is used by a particular behavior in the graph to address specific performance or robustness needs. These properties turns BITE into a flexible teamwork framework.

While these frameworks provide methods for building team oriented plans, they do not feature mechanisms for a human operator to supervise the execution of the plans, such as directing high level objectives or providing new information. In addition, while GUIs for plan development have been created [55], the BDI architecture does not inherently provide a graphical representation of the overall plan. Furthermore, there are no built-in properties of these languages which can be leveraged to build tools for validation or verification. In the next section, we present Petri Net representation which provides a graphical modeling tool for designing and analyzing team plans.

2.2 Petri Nets Plan Representation

Petri Net (PN) [52] is a mathematical and graphical modeling tool for describing concurrency and synchronization in distributed systems. It is a popular choice for designing, executing and/or monitoring multi-robot systems. Petri nets give an intuitive view of the plan. Moreover, there are several analysis methods for Petri Nets [11,45] which can test different properties, such as reachability, boundedness, liveness, reversibility, coverability and persistence. These methods allow for finding error before the testing phase on simulated or physical platforms hence, providing a significant help to the system designers.

Graphically, Petri Nets are directed bipartite graph in which nodes could be either places or transitions, arcs connect places and transitions and vice versa. Places in a PN contain a discrete number of marks called tokens. A particular allocation of tokens to places is called a marking and it defines a specific state of the system that the PN represents. Weights on the arcs define the number of tokens that must be present in certain places to trigger a change in the system, which results in token movement. This greatly simplifies representing the statuses of multiple team members and allows for a compact representation for synchronization. Formally, a PN is a tuple $PN = \langle P, T, F, W, M_0 \rangle$, where:

- $P = \{p_1, p_2, ..., p_m\}$ is a finite set of places.
- $T = \{t_1, t_2, ..., t_n\}$ is a finite set of transitions.
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs.
- $W : F \to \mathbb{N}$ is a weight function.
- $M_0 : P \to \mathbb{N}_0$ is an initial marking.
- $P \cap T = P \cap F = T \cap F = \emptyset$ and $P \cup T \cup F \neq \emptyset$

The markings of a PN evolves based on the firing behavior of the transitions. A transition t can fire whenever it is enabled (i.e., when each input place p_i of the transition is marked with at least $W(p_i, t)$ tokens) and if the transition fires $W(p_i, t)$ tokens are removed from each input places p_i and $W(t, p_j)$ tokens are added to each output place p_j.

There are several approaches proposing the use of Petri Nets for representing team plans, such as Petri Nets Plans (PNP) [80], Colored Petri Nets (CPN) [37], Task Petri Nets [73], Agent Petri Nets [43] and PrT Nets [77]. The next two sections will explain PNP and CPN in more details.

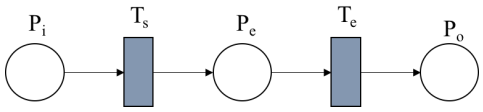
2.2.1 Petri Net Plans

Petri Net Plans (PNP) [80] take inspiration from action languages and offers a rich collection of mechanisms for dealing with action failures, concurrent actions and cooperation in a multi-robot context. The PNP is built from PN structures, actions and operators, as seen in Figure 2.1 and 2.2.

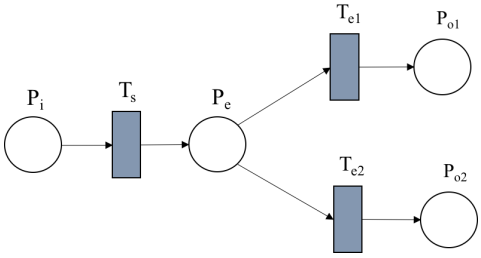
There are two types of action (or elementary structures) in PNP: ordinary and sensing actions. The usual actions, called ordinary action are common deterministic non-instantaneous actions, defined by a start event, an execution state and a terminal state. While, sensing actions are non-deterministic and the outcomes depend on some properties which may be known only at execution time. Figures 2.1(a) and 2.1(b) show these action types.

The elementary structures can be combined in series and parallel to form complex behaviors. Three main operators, sequence, concurrency and interrupt, are defined to create these complex structures, as shown in Figure 2.2.

One important functionality offered by the formalism of PNP is the possibility to modify the execution of a plan at run-time using interrupts. Figure 2.2(c) shows the structure of interrupt operator in which, the execution of the action in PNP1 is interrupted and PNP2 executes. The interrupt property of PNP framework is a powerful operation, that allows flexible recovery upon action failures.



(a) Structure of ordinary action in PNP.



(b) Structure of sensing action in PNP.

Fig. 2.1. Elementary structures in PNP.

While the use of Petri Nets allows for a centralized view of the entire team, the PNP framework includes functionality to build distributed plans for each robot from the centralized version. Different sections of a PNP correspond to activities of different robots in the team, with a token for each robot indicating its current action.

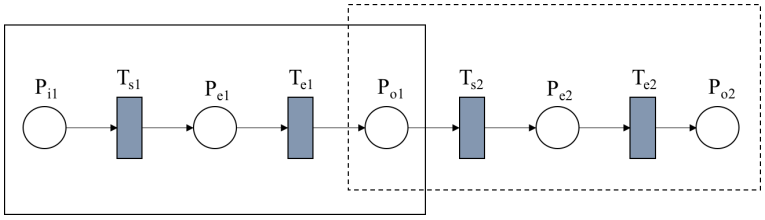
While PNP framework provides facilities for handling unexpected events, it does not explicitly consider the involvement of human operators and their intervention in case of robot failures or unexpected events. Later in section 5 we will explain an interrupt mechanism, which is based on CPN, that allows a human operator smoothly interrupt team plans to handle unexpected events.

2.2.2 Colored Petri Nets

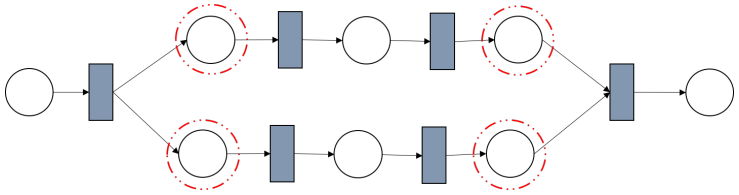
Colored Petri Nets (CPN) [37] extend Petri Nets where tokens have attached data values called the token's color. The firing behavior of transitions and consequently the evolution of markings depend on a token's color. In particular, tokens can now be identified and related to specific agents/robots, thus providing a compact and convenient modeling language for team oriented plans. In addition, transitions can modify the value(s) of the token's color when they are fired.

Similar to PN, CPN can be analyzed and verified either by means of simulation or formal analysis methods [57], thus allowing validation of team oriented plans before their execution. In more details, a CPN is defined by the tuple $CPN = \langle \Sigma, P, T, A, N, C, E, G, I \rangle$, where:

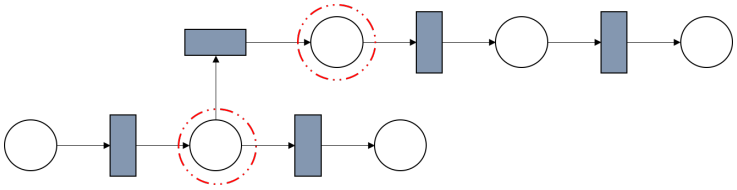
- Σ is a finite set of data types called *color sets* defined within CPN model. This set contains all possible colors, operations and functions used within CPN.
- $P = \{p_1, p_2, ..., p_m\}$ is a finite set of places.
- $T = \{t_1, t_2, ..., t_n\}$ is a finite set of transitions.
- $A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs.



(a) Structure of sequence in PNP.



(b) Structure of concurrency in PNP.



(c) Structure of interrupt in PNP.

Fig. 2.2. Operators in PNP.

- $N : F \rightarrow (P \times T) \cup (T \times P)$ is a node function.
- C is a color function that maps places in P into colors in Σ.
- E is an arc expression function. It maps each arc $a \in A$ into the expression e. The input and output types of the arc expressions must correspond to the type of the nodes the arc is connected to. Use of node function and arc expression function allows multiple arcs connect the same pair of nodes with different arc expressions.
- G is a guard function that maps each transition $t \in T$ to a guard expression g. The output of the guard expression should evaluate to Boolean value: true or false.
- I is an initialization function. It maps each place p into an initialization expression i such that $\forall p \in P : [\ Type(I(p)) = C_{MS}(p)]$ where $p(a)$ is the place of $N(a)$ and C_{MS} is all the multi-sets over C.
- $P \cap T = P \cap A = T \cap A = \emptyset$ and $P \cup T \cup A \neq \emptyset$

Definition 1. Multi-set
A multi-set m over a non-empty set S is a function $m : S \rightarrow \mathbb{N}_0$ where for a given $s \in S$ returns the number of occurrences of s in m.

In contrast to PNP [80] where, a team-plan is a collection of several single-agent plans represented with standard Petri Nets, CPN allows us to represent plans

involving several agents with a very compact structure as agents are represented by the colored tokens and not explicitly in the network. Moreover, by using CPN we can represent different types of interrupts, i.e., team-level and platform specific (see chapter 5) thus providing a rich model to allow sophisticated interactions between the human operators and team plans.

Although, PN-based approaches provide a rich modeling specification by describing the process that a robot must follow during a mission but, the decision to fire a transition is not an easy problem to solve. In other words, there are situations that the plan itself cannot fire a transition, e.g. it depends on the very specific states of a robot that can not be perceived by the high level specification (plan). So, the decision making problems here need a different approach. We focus on when robots should ask for the operator's intervention or help.

Considering robots capable of reporting their own states, such as inability to move or sensor failures, are now credible with the progress in robotic fields. Besides this self-reflection concept [63], the robots are provided with more autonomy in decision making at specific circumstances. These advancements can facilitate the development of complex multi-robot applications. In the next chapter, we will introduce the concept of the self-reflection and autonomy in robotic fields.

3

Self-Reflection and Autonomy in Human-Multi-Robot Interactions

As the size of the team grows or the demands of the environment increase (e.g. several robots need the operator's attention at the same time), the operator's monitoring and supervisory role becomes critical. Recent studies in robotic field has progressed to the point that the robots are capable of recognizing and reporting their abnormal conditions (e.g. inability to move or sensor failure). These reports will alleviate the operator's monitoring task and give him/her more time to focus on robots needing interventions and thus increase the number of robots that can be serviced over the intervening interval. However, in this scenario (i.e. self-reflecting robots) the operator cannot handle all requests at the same time. Hence, these requests will be queued and addressed sequentially. In such multi-robot applications, where the robots ask for operator's help, the cognitive workload of the operator may still be unbearable. So the main challenge is how to support the human operator and/or the robots with their decision making tasks to improve the overall system performance. The decision of how and in which order the operator should reply to a set of requests and the decision of whether or when the robots should ask for help are important examples of decision making issues. In this chapter, first we review the state of the art robotic studies where robots ask for operator's attention. Then, we present a brief review of Markov Decision Process (MDP) [7,8] and Decentralize MDP (Dec-MDP) [10, 30] as the basis for decision making under uncertainty. Next, we discuss Reinforcement Learning (RL) [68] approaches and particularly Q-Learning [76] because they are popular frameworks for solving MDPs when the dynamics of the environment are unknown. Afterwards, we present a brief introduction to the Balking Queue [46] structure in which, the arrivals can choose to join the queue or not.

3.1 Self-Reflection in Robotic Applications

The concept of self-reflection [63] refers to the (limited) capability of robots in reflecting the demands for attention, permission or any types of assistance in robotic applications. We concern the works in which, the robots can perceive their situations and inform or ask the operator for help.

Some researches in this area consider collaborative control multi-robot scenario to address the limitation of the robots. In these models [28, 29], the human is presumed as a teammate for the robot and provides the robot with extra information whenever they ask for his/her opinion.

Authors in [59] use this idea to make a service robot overcomes its lack of capability of doing certain tasks. For example, the service robot without a manipulator cannot press the elevator's button. Thus, it looks for human helps and must wait for the human. However, human(s) in their application are not assigned to a supervisory role or responsible to help the robots.

Bevacqua et. al [13] have proposed a framework which allows a single operator to interact with a set of UAVs by utilizing natural mixed initiative communication. However, in their model, the human operator is also involved in the scene and is thus co-located with the robots. As a result, the human operator cannot be fully dedicated to the robotic platforms, and can only provide sparse and sketchy commands. In a similar direction, authors in [66] propose a leader-follower approach for collaborative object manipulation where the human (leader) is considered as part of the team while controlling the team with the movements of his/her hand. In these works the operator is not only a supervisor but also involved in performing other tasks (i.e. as a teammate for other robots, so they need to coordinate) In the above examples, the human can become overwhelmed by increasing the number of requests. To address this challenge, some works consider providing the human with intelligent interfaces. For example, Stocia et al. [67] provide a human-friendly gesture-based system which maps human inputs into robotic commands. Authors in [31] consider a complete multi-modal interaction prototype which supports the human with speech, arm and hand gestures to select, localize and communicate to one or more robots in a search and rescue mission.

Some other works try to decrease the operator's (cognitive) workload by providing decision making solutions to the operator. For example, Authors in [58] propose a software agent to assist the human. In their model, the agent prioritizes the tasks of the human and suggests him/her what should be done next. Then the human operator decides either to follow the advice or not. The paper shows the improvement in the team's performance, though, the robots should wait for the operator's reply (usually in a passive mode).

The concept of Adjustable Autonomy or mixed initiative has been the basis of many research in the field of human-multi-robot interaction. In more details, Adjustable Autonomy defines the level of autonomy of the robots where robots can vary their level of autonomy and transfer decision-making control to humans (or other agents) [22,62,69]. The key issue in this setting is to devise effective techniques to decide whether a transfer of control should occur and when this should happen. Different techniques have been proposed to address this challeng, for example, [34] consider that the robot will ask for human help/intervention when the expected utility of doing so is higher than performing the task autonomously, or when the uncertainty of the autonomous decision is high [19, 33] or when the autonomous decision can cause significant harm [24]. However these decision making solutions usually have been considered as individual one-shot or one-step decisions without considering the long-term cost or the cost of the decisions on the other team members (if any).

Scerri and colleagues [62] propose the use of transfer of control strategies which are conditional sequences of two types of actions: transfer of decision making control (e.g., an agent giving control to a user) and coordination changes (i.e., an agent delaying the execution of a joint task). The authors propose an approach based on Markov Decision Processes to select an optimal strategy and evaluate their method in a deployed Multi-Agent System where autonomous agents assist a group of people in daily activities (e.g., scheduling and re-scheduling meetings, ordering meals, and so forth).

In this thesis, our focus is on the supervisory role of human operator for handling unexpected events in multi-robot systems. Particularly, we are looking at multi-robot applications where the robots can communicate their need for interaction to the operator and very likely the human cannot handle all requests at the same time, hence, the resulting human-multi-robot interactions would form a queuing system. The idea of applying queue theory to multi-agent systems to improve the supervisory role of operators has been studied in the literature [20, 42]. For example, [20, 42] examine and compare first-in-first-out (FIFO) and shortest-job-first (SJF) queuing models where the requests will be queued according to their arrival time and shortest service time respectively. In another experiments [20, 42], they show that, in a first-in-first-out (FIFO) queue displaying a single request at a time led to poorer performance than one showing the entire (Open) queue. In summary, the focus of such previous works was on investigating different queue disciplines (i.e. the order in which the requests should be processed by the operator). Since, the autonomy of robots was not addressed in those work, the queue size may grow indefinitely as no robot will leave the queue before receiving the operator's attention. This will impact team performance, since it can significantly delay the operations of robots waiting in the queue.

To address this issue, here we consider a particular queuing mechanism, *Balking Queue*, in which the autonomous agents can decide whether to join the queue or act autonomously based on some key information, such as the severity of the request or the number of requests inside the queue. We explain the basic idea of *Balking Queue* in Section 3.4. We present our contribution to this queuing system an applying it to robotic applications in Chapter 6.

3.2 Markov Decision Process

The decision of whether to join the queue or not for each situation of each robot will impact the future decisions of the corresponding robot and the other robots. As a result, each robot should be provided with a sequential decision making framework that considers the long-term affect of each action (balk or join) on the future state of the system under uncertainties. we are concerned here with sequential decision problems, in which the team's utility depends on a sequence of decisions. Markov Decision Process (MDP) provides a mathematical framework for modeling sequential decision making problems under uncertainty [8, 12, 35, 53]. The main components of an MDP are the following:

- S is a finite set of states.
- A is a finite set of actions.

- $T(s\prime \mid s, a) = P[S_{t+1} = s\prime \mid S_t = s, A_t = a]$ is a state transition probability function. The transition probability only depends on the previous state s and the action a, which is called the Markov condition.
- $R(s, a) = E[R_{t+1} \mid S_t = s, A_t = a]$ is a reward function. In other words, R provides feedback from the environment.
- H is the horizon over which the agent will act that can be finite or infinite. In general, the horizon H shows how much into the future we consider for optimizing the expected reward. Finite horizon problems consider a fixed, predetermined number of time steps to maximize the expected reward. However, in infinite horizon problems, the steps could vary and could be infinite. We will focus on the latter, where the horizon of our model is the length of a team mission.

The task of deciding which action to choose in each state is done by a policy function $\pi(s)$. In other words, a policy in MDP, $\pi(s)$, is a mapping from states in S to actions in A in such a way that it will optimize some objective function (e.g. it will maximize the expected total discounted reward). The value function $V^{\pi}(s)$ gives the long-term value of state s under the policy π as follow:

$$\forall s \in S : V^{\pi}(s) = E_{\pi}\left\{ \sum_{h=0}^{H} \lambda^h r_{h+t+1} \mid s_t = s, \pi \right\} \tag{3.1}$$

where $0 \leq \lambda < 1$ is the discount factor (for infinite horizon) and t is the decision epoch.

Or similarly, the value of taking action a in state s under a policy π, denoted by $Q^{\pi}(s, a)$, as the expected return starting from s, taking action a, and following policy π:

$$Q^{\pi}(s, a) = E_{\pi}\left\{ \sum_{h=0}^{H} \lambda^h r_{h+t+1} \mid s_t = s, a_t = a, \pi \right\} \tag{3.2}$$

This function is usually know as Q-function and the corresponding values as Q-values.

In other words, we are looking for the optimal policy as:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} E\left\{ \sum_{h=0}^{H} \lambda^h R_t(S_t, A_t, S_{t+1}) \mid \pi \right\} \tag{3.3}$$

The term Dynamic Programming [8, 35] refers to a class of algorithms that is intended to find optimal policies in the presence of a model of the environment (i.e. the dynamics T and the rewards R) which, in general could be unknown. Moreover, the main limitation of dynamic programming methods are the explosion of memory and the time requirements to find the optimal policy, when the problem size grows.

Different techniques have been developed for MDP that focus on the feasibility and efficiency of the problem solution [5, 6, 9, 17, 30, 32, 32, 44, 48, 54, 65]. On the other hands, many real world problems can be decomposed into distributed subproblems, each could be solved by an agent to tackle the exponential state and action spaces.

In the next section, we will explain Decentralized MDP (Dec-MDP) in more details.

3.2.1 Decentralized Markov Decision Process

Multi-robot coordination problems are often formalized with Markovian models. These models allow to represent situations where multiple robotic agents aim at optimizing a shared reward function and are designed to take uncertainty into account. In more detail, when agents have access to the full and complete state of the world (directly or through communications), the problem can be modeled as a Multi-agent Markov Decision Process (MMDP) [15]. However, in many real world applications due to the limited, costly or unavailable communications, accessing to the full state of the environment and the state of other agents is difficult.

Thus, in such applications, decentralized decision making solutions are often preferred to the centralized one. In more details, a Dec-MDP is defined by a tuple $\langle S, A, P, R \rangle$ where:

- S is the set of world states which is factored into $n+1$ components, $S = S_1 \times ... \times S_n$. In a special case (a.k.a Factored n-agent Dec-MDP), S_i refers to the local state of agent i. In Dec-MDP, the state is jointly fully observable which means that the aggregated observations made by all agents determines the global state.
- A_i is the set of actions for agent i where $A = \times_i A_i$ is the set of joint actions.
- $P = S \times A \times S \rightarrow [0, 1]$ is the state transition probability.
- R_i is the immediate reward obtained by agent i, taking action a_i in state s_i.

This model suffers from exponential state space and joint-action space which makes it intractable [1]. In more details, the complexity of Dec-MDP is nondeterministic exponential (NEXP) hard, even when only two agents are involved [10]. The following properties of Dec-MDP models are the two most significant assumptions that match real world domains by reducing the complexity of the problems [1, 4–6]:

Transition Independent A (factored) n-agent Dec-MDP is said to be *transition independent* if the state transition probabilities factorize as follows:

$$P(s\prime \mid s, \vec{a}) = \prod_i P_i(s_i\prime \mid s_i, a_i) \tag{3.4}$$

where, $\vec{a}$ is the joint actions and $P_i(s_i\prime \mid s_i, a_i)$ represents the probability that the local state of agent i transitions from s_i to $s_i\prime$ after executing action a_i.

Reward Independent A (factored) n-agent Dec-MDP is said to be *reward independent* if there exist R_1 through R_n such that $R(s, \vec{a}) = \sum_i R_i(s_i, a_i)$. In other words, the overall reward is composed of the sum of the local rewards where, each local reward depends only on the local state and action of one of the agents.

3.3 Multi-Agent Reinforcement Learning

Reinforcement Learning (RL) [14, 16, 68, 71] approaches are commonly used in robotic systems where the dynamics of the environment are often unknown. The aim of reinforcement learning approaches is getting an agent to interact with the environment (i.e. the MDP), to find the optimal behavior (policy), being guided by the evaluative feedback (rewards).

There are two groups of RL methods: model-free and model-based methods. Model-based RL approaches are preferred in the environments where performing many real simulations are impossible or the robot will break. Therefore, these methods first try to learn the model of the environment and then solve the problem. However, in our work we have access to a simulation environment that we can generate many samples. Hence, we concentrate on model-free approaches. Two of the leading approaches for model-free reinforcement learning are Q-learning [76] and policy gradient methods [68].

Q-learning approach keeps a table for each state and action while policy gradient methods are useful for high-dimensional or continuous action spaces. Q-learning method is widely used due to its simplicity and good real time performance. It is an off-policy method which learns optimal Q-values, and determines an optimal policy for the MDP. The overall idea of off-policy methods is to evaluate or improve a policy different from that used to generate the data [68] while on-policy methods attempt to evaluate or improve the policy that is used to make decisions.

Since we are using Reinforcement Learning in a multi-robot application, it is important to have a brief review on the challenges in Multi-Agent Reinforcement Learning (MARL). The main challenges in cooperative [1] MARL include: the non-stationarity of learning environment and credit assignments.

The first challenge, non-stationary environment, happens due to the fact that all the agents learn simultaneously. When the agents learn, they modify their behaviors, which in turn can change other agents' learned behaviors [49]. To deal with this issue, one plain approach is to assume the other learners as part of the dynamic of the environment. This approach is controversial, since the agents are co-adapting to each other's behavior which may change the environment itself [49, 64, 79]. Later in Chapter 6 we will discuss that this issue has less impact on the specific problem we are modeling.

The credit assignment brings the problem of how to divide the reward, received through the joint actions, among agents at each decision epochs. *Global reward* solution considers dividing the received rewards equally among agents while *local reward* solution rewards the learners for the actions they take. In distributed environments, which is usually the case, the *global reward* approach does not work well. Author in [2] investigates different credit assignment approaches. The experiments show that, local reward leads to faster learning rates. For one problem (foraging), local reward produces better results, while in another (soccer) global reward is better. The author suggests that using local reward increases the homogeneity

[1] Our focus is on cooperative multi-robot teams where the robots are trying to maximize a shared utility which is opposed to competitive multi-robot applications.

of the learned teams. This in turn suggests that the choice of credit assignment approach should depend on the specific problem [49].

In this section, we provide a brief preliminary background on Q-Learning which is the basis of our learning models in Chapter 6.

In a general setting, the robot interacts with the environment (i.e. selects an action), receives the immediate reward and updates its state-action values (i.e. Q-values) according to (3.5):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_{a\prime \in A} Q(s_{t+1}, a\prime) - Q(s_t, a_t)) \tag{3.5}$$

where R_{t+1} and s_{t+1} are respectively the reward and the state observed after performing action a_t in state s_t; $a\prime$ is the action in state s_{t+1} that maximizes the future expected rewards; α is the learning rate and γ is the discount factor.

One of the challenges in reinforcement learning is the trade-off between exploration and exploitation. To obtain more rewards, a robot must prefer actions that it has tried in the past and found them effective (i.e. exploitation). But to discover such actions, it has to try actions that have not been selected before (i.e. exploration). In Q-Learning there are different methods for action selection such as ϵ*_greedy* and *softmax* to balance exploration/exploitation:

ϵ_greedy parameter ϵ determines the randomness in action selections such that with probability ϵ, the action with maximal estimated action value will be selected and with probability $1-\epsilon$ a random action will be selected. That is, all non-greedy actions has the same probability of being selected.

softmax Bias exploration towards promising actions. In other words, *softmax* method varies the action probabilities as a graded function of estimated value. The greedy action is still given the highest selection probability, but all the others are ranked according to their value estimates.

MARL:

3.4 Balking Queue Model

The first mathematical model of a queuing system with rational customers was formulated by Naor [46]. In his model, customers upon their arrival decide according to a threshold value whether to join the queue or not (balk). The individual's optimizing strategy is straightforward, a customer will join the queue while n other customers are already in the system if

$$R - nC\frac{1}{\mu} \geq 0 \tag{3.6}$$

where a uniform cost C for staying in the queue and a similar reward R for receiving service are assigned to each user and μ is the intensity parameter of exponentially distributed service time.

Thus, $n = \lfloor \frac{R\mu}{C} \rfloor$ serves as a threshold value for balking, that is if the number of customers waiting in the queue is greater than n, the newly arrived customer will

not join the queue. For computing the threshold value, the model in [46] assigns a generic reward associated with receiving the service and a cost for waiting in the queue to each customer, but there is no gain or loss associated with the balking action.

The main benefit of using this queuing structure in a multi-robot application is that, an appropriate balking threshold can minimize the waiting time of the system. In other words, a rational agent prefers not to send its request to the operator (i.e. queue) when it finds the queue size to be too long. This will avoid increasing the queue length while increasing the chance of future request to be send to the queue. This threshold and decision must be computed carefully, taking into account the waiting cost, the importance of the current request (i.e. reward) and the future requests.

However, when applying this model [46] to a robotic application, there is no clear indication how such a threshold can be computed and essentially this model does not consider the cost of balking. Our focus is on showing how the elements (i.e. reward and cost) of balking strategy should be adjusted according to a practical robotics scenario. In other words, to apply this model to a robotic application, different rewards and costs for each type of request should be considered. For example, a robot with a high severity request cannot balk the queue only because the queue is too long. Moreover, balking has a cost for the team since the robot that balks a request may not be able to accomplish its assigned task(s). In this case, the remaining task(s) of that robot should be reassigned to the other robots which brings an extra load to both the operator and other team members. In order to adjust the balking model to work with a real environment considering all above elements, one convenient and practical technique is reinforcement learning where the robots can learn the balking policies through direct interaction with the environment. We present and compare different models for balking policies in Chapter 6.

4

Motivating Domain: Cooperative Water Monitoring Application

This work focuses on a system of robotic boats developed as part of the Cooperative Robotic Watercraft (CRW) [60] and IntCatch project[1]. The project establishes a novel approach for monitoring and management of river and lake water quality. The aim is to improve the performance and decrease the cost of monitoring making use of innovative technologies and user friendly platforms. The motivations for choosing this framework as our research domain include: first, relative to other types of vehicles, watercraft are inexpensive, simple, robust and reliable. Specifically, they are low cost air-boats that use an above-water fan to propel themselves forward safely and effectively through shallow or debris-filled water. Thus, many applications can be done through a team of these air-boats. Second reason for choosing this framework is, because the framework provides a close interaction solution between human users and autonomous platforms, hence making this application more convenient for citizen science and community engagement. All these characteristics make this application an interesting research domain for our work. In this section we provide a brief overview of the whole system and we describe the team plan specification language used in the system.

4.1 The Cooperative Robotic Watercraft System

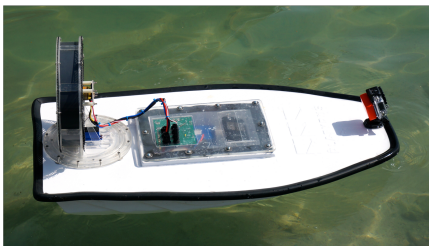
Figure4.1(a) shows a robotic air-boat. In addition to a battery based propulsion mechanism, each boat is equipped with an Android OS smartphone, custom electronics board, and sensor payload. The Android smartphone provides communication, either through a wireless local area network or 3G cellular network, GPS, compass, and multi-core processor. An optional prism can be mounted to the transparent lid of the waterproof electronics bay to use the phone's camera for stationary obstacle avoidance and imaging.

The Arduino Mega based electronics board receives commands from the Android phone over USB OTG and interfaces with the propulsion mechanism and sensor payload, as shown in Figure 4.1(b). The electronics board supports a wide variety of devices including acoustic doppler current profilers and sensors that mea-

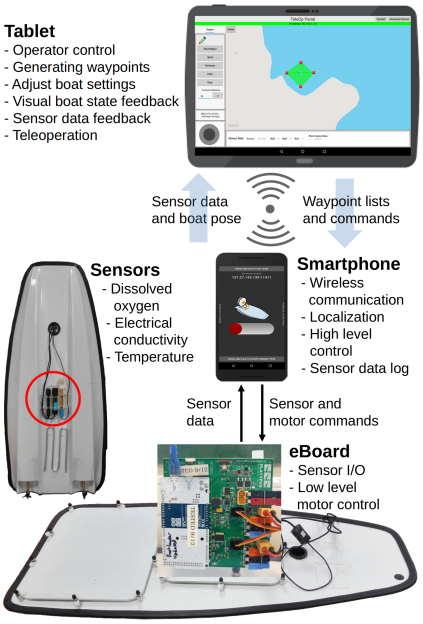
[1] `http://intcatch.eu/index.php`

sure electroconductivity, temperature, dissolved oxygen, and pH level. All sensor data is logged with time and location.

The robot team is controlled from a nearby base station via a high power wireless antenna or remotely using 3G connectivity. The operator uses a SAMI compatible GUI to instantiate SPN plans, monitor their execution, and provide input as necessary. In this case, compatibility means the GUI contains a library of UI components listing which data classes and SAMI markup they support, allowing a custom "interaction panel" to be constructed for each event requiring operator input.



(a) The robotic boat equiped with innovative sensors for the online measurment of important parametes in water bodies.



(b) System architecture

Fig. 4.1. A robotic platform and a diagram of the system architecture.

4.2 Supervisory Framework: SAMI

The supervisory framework of our water monitoring application, SAMI Petri Nets (SPN), is based on Colored Petri Nets and Hierarchical Petri Nets, with several extensions to add the capability to send and receive commands and information from team members, to perform and reference task allocations, and to capture situational awareness and mixed initiative (SAMI) directives. We define an SPN structure as the following tuple $\langle P, T, F, E, R, SM \rangle$, where:

- $P = \{p_1, p_2, ..., p_i\}$ is a finite set of *places*.
- $T = \{t_1, t_2, ..., t_j\}$ is a finite set of *transitions*.
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of *edges*.
- $E = \{e_1, e_2, ..., e_k\}$ is a set of *events*.
- $R = F \rightarrow \{r_1, r_2, ..., r_m\}$ is a mapping of *edges* to a set of *edge requirements*.
- $SM = P \rightarrow \{sm_1, sm_2, ..., sm_l\}$ is a mapping of *places* to a set of *sub-missions*.

The SPN models the execution of a team plan by representing the current state of the system (i.e., the markings of the places), the evolution of system states over time, and the interactions between the different components of the systems. In more detail, the SPN implementation defines a *plan manager*, which is an execution engine responsible for all interactions among the different components of the robotic platforms. All the interactions take the form of commands (or requests) sent from the plan manager to the robotic platforms (or to the operators) and information received from human operators/robotic platforms.

While we will use the SPN framework in our application domain (i.e., cooperative robotic watercraft), we make no context specific assumptions for the team plan specification language (and for the interrupt mechanism we will define in Chapter 5. Hence our approach can be used in other scenarios where human operators should design and monitor team plans for multi-robot systems.

In what follows, we describe each of the main elements of the SPN and then provide operational semantics in the form of firing rules for the transitions.
Events fall under two categories: *output events* and *input events*.
Output events are associated to places in the Petri Net (using the mapping $E_O = P \rightarrow \{oe_1, oe_2, ..., oe_k \subseteq E\}$, which maps each place to a set of output events) and represent commands or requests that are sent to human operators, robot proxies[2], or agents. When a token(s) enters a place, all the output events on the place, $E_O(p)$, are processed. The registered handler for that class of output event is sent the output event *oe* along with the tokens that just entered the place (Algorithm 3).

For output event classes that contain data fields, there are 3 ways to specify the information, which are listed here with example usage in our outlined scenario: (1) Value defined offline by the Petri Net developer (the battery voltage threshold to send a low-energy alert to the operator). (2) Value defined by the operator at run-time (a safe temporary position for robots to move to in order to avoid an incoming manned boat). (3) Variable name whose value is written by an input event at run-time (a variable to retrieve the path returned from a path planning agent via a "Path Planning Response" input event). Variables are explained in more depth later in this section.

Input events are associated to transitions (using the mapping $E_I = T \rightarrow \{ie_1, ie_2, ..., ie_h \subseteq E\}$, which maps each transition to a set of input events) and contain information received from human operators, robot proxies, or agent services, which perform assistive functions such as path planning, task allocation, and image processing. The set of input events on a transition, $E_I(t)$, are responses to an output event on a place preceding the transition. For an input event *ie* that

[2] With the term *proxy* we refer to a software-service that connects a specific boat with the rest of the system

will contain information at run-time (such as a generated path or selection from an operator), a variable name is used so the information can be accessed by output events.

Input events contain "relevant proxy" and "relevant task" fields, which contain the identities of the proxy(s) or task(s) (if any) that sent or triggered the input event.

Events in SPN have a function that is very similar to actions in the PNP framework [80]: the PNP framework describes the evolution of a robotic system where states change due to actions and SPN describes the evolution of a team plan where the states change due to events. However, an important structural difference is that in PNP actions are associated to transitions, while in SPN we associate output events to places and input events to transitions. The rationale behind this choice is twofold: first, we have a more compact SPN, second, this results in a more efficient implementation. To see this, consider the place with output event "ProxyExecutePath" in Figure 4.2 which is connected to a transition with "ProxyPathCompleted". This path execution sequence is captured with one place and one transition. If we instead associate output events with transitions, we would need a place representing the precondition for starting ProxyExecutePath, a transition that actually sends the ProxyExecutePath, a second place that represents that the proxies are executing the path, and a second transition with the ProxyPathCompleted input event. This extra place and transition for each action sequence results in a much less compact network. In addition, we use the output event instance's unique id as criteria for matching a received input event to a transition in the SPN. This is necessary in the common case where an input event is used in multiple transitions, such as having instances of ProxyExecutePath and ProxyPathCompleted, so that the correct transition's firing requirements are updated. In contrast, associating output events to transitions would make the pre-conditions and post-conditions for the events more visible in the CPN representation. This could be a valuable feature for a designer and would be more in line with traditional PN specifications of control systems. However, a precise assessment of this trade-off requires further investigations while our focus here is to provide a mechanism for smoothly handling interrupts. Hence, we leave the analysis of this issue as a future work.

When an input event is received by the system and matched to its corresponding transition in a Petri Net, it is marked as being "received" (Algorithm 1). When a transition fires, its input events' "received" statuses are reset (Algorithm 2).

Variables Similar to the model for CPN [39], SPNs support a variable database, where variables are typed and scoped globally or locally. The use of variables is a key element to keep the network compact and to make the plan specification framework flexible and easy to use. Global scope variables allow plans to share information, such as a sensor mapping density, while local scope variables allow multiple copies of a plan to run simultaneously without overwriting instance specific data, such as locations to visit. Different variables can be defined for each input event. Fields in output events can refer to these variables, provided they are

of the corresponding type and within scope.

Tokens In general CPN modeling language allows defining a variety of color sets for tokens in order to support different data types such as list, structure, enumeration, etc. We now explains our data types in SPN. The SPN tokens have four pieces of information: a name (String), a token type (TokenType), a proxy (ProxyInt), and a task (Task). Each token tk is one of three TokenTypes: *Generic* tokens have no defined proxy nor task and are used as counters. *Proxy* tokens contain a proxy but no task. These are created whenever a robot proxy is added to the system at run-time. *Task* tokens contain a task and might contain a proxy. Task tokens are created by the Petri Net execution engine when a plan is started, creating one for each task in the plan. When the task is allocated to a proxy, the proxy field of the task's corresponding token is set to the proxy assigned to the task. Representing proxies and tasks using tokens allows for multi-robot plans with arbitrary numbers of team members to be constructed and visualized compactly, compared to having an individual Petri Net for each member of the team.

Edge Requirements *Edges* fall under two categories: *incoming edges* $if \in F$, which connect a place to a transition, and *outgoing edges* $of \in F$, which connect a transition to a place. Similarly, *Edge Requirements* have two categories: *incoming requirements* ir, which are mapped by R from *incoming edges*, and *outgoing requirements* or, which are mapped by R from *outgoing edges*. In a standard Petri Net, incoming edges have a weight which specifies the number of tokens required for a transition to fire, which are then consumed, and outgoing edges have weights which specify the number of tokens to add to the connected place.

Colored Petri Nets allow edges to specify different quantities for different colors of tokens. SPN edge requirements have additional options to maintain the network as compact as possible.

Each **incoming requirement** ir on an *incoming edge* if, $R(if)$, specifies tokens that must be present or absent in the connected place in order for the connected transition to fire. However, when a transition fires, these tokens are not removed as it could cause undesired interruption of behavior controlled by output events in the connected place. Instead, each **outgoing requirement** or on an *outgoing edge* of, $R(of)$, specifies tokens that should be removed from the incoming places (the places preceding the connected transition) and tokens that should be added to the connected place.

This is achieved by having each outgoing requirement specify a set of tokens and an action to perform on those tokens: take, consume, or add. Taking a token removes it from incoming places and adds it to the outgoing place. Consuming a token removes it from incoming places. Adding a token adds a copy of the token to the connected place. The take action represents the standard operation that is executed on PN and CPN when a transition fires. However, *consume* and *add* are extensions to the standard semantics of PN used in SPN only to maintain the network's compactness. Specifically, the motivation for using these actions is that since we have output events associated to places, we need a way to move a token from a preceding place to a following place without removing it from the initial place. If we expand the network as described above (i.e., adding two places and

one transition) we would not need this extension. Furthermore, while we could use standard PN structures to implement these actions (e.g., we could add a specific transition without outgoing edges to consume a token from a place) this would defeat the purpose of having a compact network.

Similar to Colored Petri Nets, the set of tokens specified by an edge requirement can be generic tokens or specific task tokens. Edge requirements can also refer to "relevant tokens" which are defined by the input events on the transition being evaluated. The list could contain proxy token(s), in the case of a "Path Completed" input event which specifies the proxy token for the robot that finished, so that at run-time that proxy token can be moved forward in the Petri Net. It could also contain task token(s), in the case of a "Task Completed" input event signaling that a particular task has been completed.

Sub-missions The SPN language supports hierarchical team plans, allowing a place (called a *sub-mission place*) to have a set of "sub-missions", $SM(p)$. Each sub-mission sm is an SPN which is run in either *dynamic* or *static* mode. For **dynamic** sub-missions, when tokens enter the *sub-mission place* of the parent plan a new instance of the sub-mission SPN is started and the initial marking is defined as those tokens in the sub-mission's start place (Algorithm 3).
In contrast, **static** sub-missions are instantiated only once, when the parent plan is instantiated, and have an empty initial marking. They share their start place with the parent plan: tokens that enter the parent *sub-mission place* are also added to the start place of the sub-mission.
All sub-missions can return values and tokens as well as write to variables shared with their parent plan. When a token(s) enters an end place in a sub-mission, the sub-mission is marked as being "complete." Until then, transitions in the parent plan leaving the sub-mission place are prevented from firing (Algorithm 1). When a transition fires, the completion status of any sub-mission in an incoming place is reset (Algorithm 2). Sub-missions allow developers to reduce repeated creation of common sequences and increase readability of the plan.

Markup Each event e has a set of markup (using the mapping $MK = E \rightarrow \{mk_1, mk_2, ..., mk_n\}$, which maps each event to a set of markup). Markup are context clues associated to events which can provide several types of information: which GUI components and widgets are most appropriate for operator interaction, which set of priorities an agent service should consider when choosing from multiple algorithms, and which level of mixed-initiative autonomy to employ in making decisions.

Markups are an addition to the standard CPN that can be exploited to support situational awareness and mixed initiative control, making the model more flexible.

Each *markup* $m \in MK(e)$ has a number of options and variables that the SPN developer must specify. GUI components and agent services correspondingly indicate which markup options they support, allowing the most appropriate ones to be retrieved automatically at run-time.

For example, the "relevant proxy" markup indicates to the GUI that the locational data of certain proxies should be displayed to the operator in addition to any other information contained within the event. Settings include the proxy

selection criteria (the event's relevant proxies or all proxies) and which data to visualize (including pose, current path, future paths, and past paths). The "mixed initiative trigger" markup is used to indicate when system autonomy should make a decision and if the operator should be informed. Options range from never using system autonomy, using autonomy after a timer expires, or using autonomy immediately without consulting the operator.

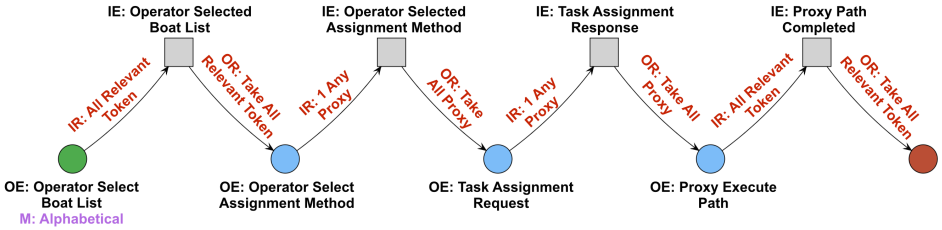
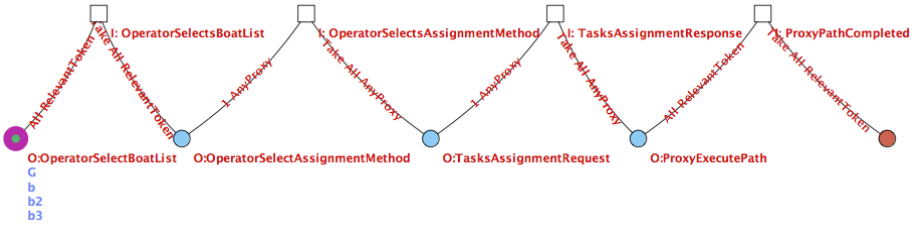


Fig. 4.2. SAMI Petri Net "Cooperative Location Visit" (CLV) plan (without the interrupts), where the operator selects a group of boats to visit a set of locations to perform point measuring tasks. The boats should navigate to each location and acquire a specific measure (e.g., pH level, oxygen level, temperature). The starting place is colored green and the end place is colored red

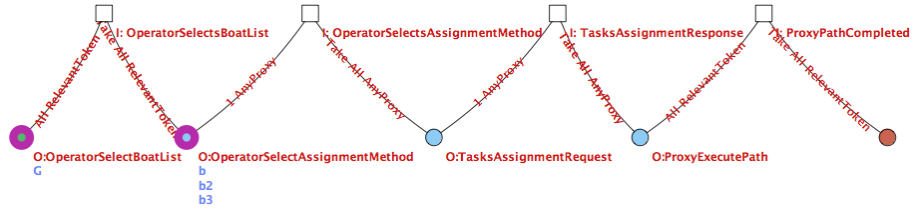
The main components of an SPN are illustrated in a sample plan in Figure 4.2. When a plan is selected to run, an initial marking is applied to the plan's start place, $p_S \in P$ (the leftmost place, colored green). When a token enters an end place, $P_E \subset P, p_S \notin P_E$, the plan terminates (the rightmost place, colored red). The initial marking is a generic token and a proxy token for each boat, which triggers Algorithm 3 when applied to p_S.

Operator Select Robot List is triggered asking the operator to select the boats that will participate in the plan from the list of corresponding proxy tokens it received. When the operator performs this action, an *Operator Selected Boat List* input event will be generated and matched to its transition in the SPN. Its received status is set to true and Algorithm 1 will be called. The transition will be enabled and fired via Algorithm 2, taking the *relevant tokens* (i.e., the tokens corresponding to the selected boat proxies) to the next place. The plan progresses in a similar way until the tokens reach the last place (i.e., all selected boats have completed their path). When this happens the plan reaches the end place and is no longer active.

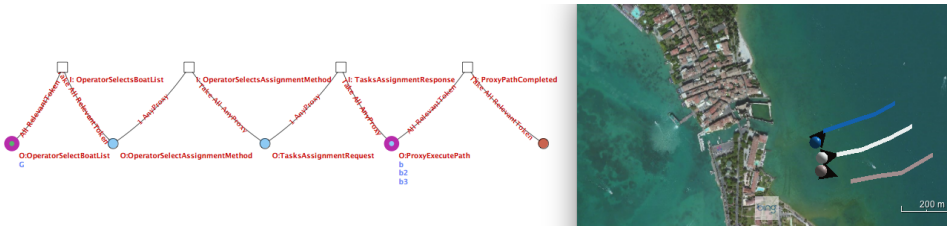
To illustrate how the concept of color is used for modeling multi robot team in SAMI platform, two consecutive markings of CLV plan execution are shown in figures 4.3(a) and 4.3(b). The upper-left corner of both figures displays the same petri net as shown in figure 4.2. These markings illustrate how the colored tokens (related to different boats) are passing through the net. In contrast to a plan created by non-colored PN, for each boat one needs to create a petri net plan and then synchronizes and connects them to make the team plan.



(a) Initial marking of the CLV plan, with 3 tokens (associated to boats) and 1 generic token(a generic token is always included in initial SPN marking to start the plan).



(b) The marking shows the state of the net after that the operator selects the boats for executing the mission. At this point 3 boats are shifted to the next place in the petri net.



(c) This marking shows the state of the net when the boats are inside the ProxyExecutePath place. At this point each boat will execute its related path based on the task assignment algorithm which is selected by the operator.

Fig. 4.3. SAMI Petri Net "CLV" plan.

4.3 Assisted Plan Design and Analysis for SAMI

In order to assist the SPN developer, we created an intelligent plan editing tool. The editor was designed with two potential limitations of the plan language in mind: overwhelming visual clutter and developer errors resulting in an invalid SPN or unexpected run-time behavior. The editor contains different visualization modes which selectively hide and compress sections of nets based on different tasks the developer may be performing. "Assistant agents" check for violations of SPN rules and flag errors, such as incomplete graphs and unlabeled start/end places, and warnings, such as suspicious edge labels.

In addition to these checks that verify syntactic properties of the SPN, we can consider typical properties for PN and CPN such as the liveness and home properties [38].

Algorithm 1 Checks if a transition should be enabled

```
1: procedure CHECK TRANSITION
2:     for ie ∈ E_I(t) do                ▷ Check that all input events have been received
3:         if ie.received == false then return false
4:         end if
5:     end for
6:     for if ∈ t.inEdges do             ▷ Check that all incoming edge's in requirements have been satisfied
7:         for ir ∈ R(if) do
8:             if ir.satisfied == false then return false
9:             end if
10:        end for
11:        p = if.start
12:        for sm ∈ SM(p) do ▷ Check that any sub-missions on an incoming place are at a goal state
13:            if sm.complete == false then return false
14:            end if
15:        end for
16:    end for
17: return true
18: end procedure
```

In more detail, as discussed in [80], some properties are particularly interesting for plan monitoring frameworks. Specifically, in [80] the authors state that a PNP must be minimal (i.e., all transitions can be fired at least once), effective (i.e., the goal marking is a home state), and safe (i.e., the Petri Net is 1-bounded). For what concerns our framework, SPNs that specify valid team plans should also be minimal and effective. Specifically, SPN should be minimal as they should not contain transitions that will never fire. Moreover, SPNs should be effective, because they encode team plans and as such they explicitly have a goal marking that must be reachable from all possible markings of the SPN (i.e., the goal marking should be a home state). However, the safety property does not apply to our framework. PNP tokens define execution threads for atomic actions, hence there should not be two tokens in the same place. In contrast, a SPN place could have many tokens, and the tokens are not necessarily of different colors; several tokens of the same color set (for example, proxy tokens for Boat A) could exist in the same place simultaneously. A motivation for this would be knowing how many times a particular proxy has triggered a contingency behavior by counting the number of proxy tokens for that proxy which are in a particular place.

The above described properties (i.e., an SPN being minimal and effective) can be checked with standard reachability analysis performed on CPN. However, this requires to transform SPN plans to standard CPN (e.g., by removing output events from places and by associating them to new transitions, as mentioned above). We performed this analysis on the plans we consider here using CPNTool [38] and our analysis reports that all plans we consider are both effective and minimal. Nonetheless, this does not imply that all SPN plans can be directly translated to an equivalent CPN and analysed using CPN Tools. This would require further investigations which fall outside the scope of the current contribution.

Algorithm 2 Fires an enabled transition

```
1: procedure FIRE TRANSITION
2:     t ∈ T                                          ▷ t is the transition we are executing
3:     TK_A = ∅ ▷ TK_A is a map associating tokens to add to outgoing places (initially empty)
4:     TK_R = ∅          ▷ TK_R is a map associating tokens to remove to incoming places (initially empty)
5:     for of ∈ t.outEdges do                         ▷ Fill in TK_A and TK_R
6:         for or ∈ R(of) do
7:             for p ∈ t.outPlaces do
8:                 TK_A.put(p, getTokensToAdd(or))
9:             end for
10:            for p ∈ t.inPlaces do
11:                TK_R.put(p, getTokensToRemove(or))
12:            end for
13:        end for
14:    end for
15:    for p ∈ t.outPlaces do
16:        enterPlace(p, TK_A(p))
17:    end for
18:    for p ∈ t.inPlaces do
19:        leavePlace(p, TK_R(p))
20:    end for
21:    for p ∈ t.inPlaces do ▷ Reset completion status of all sub-missions on incoming places
22:        for sm ∈ SM(p) do
23:            sm.complete = false
24:        end for
25:    end for
26:    for ie ∈ E_I(t) do       ▷ Reset receipt status of all input events on the transition
27:        ie.received = false
28:    end for
29:    T_check = ∅ ▷ T_check is a list of transitions we could have affected and should now check (initially empty)
30:    for p ∈ t.outPlaces do                         ▷ Fill in T_check
31:        for t2 ∈ p.outTransitions do
32:            if t2 ∉ T_check then
33:                t2 → T_check
34:            end if
35:        end for
36:    end for
37:    for p ∈ t.inPlaces do
38:        for t2 ∈ p.outTransitions do
39:            if t2 ∉ T_check then
40:                t2 → T_check
41:            end if
42:        end for
43:    end for
44:    for t2 ∈ T_check do
45:        if checkTransition(t2) == true then
46:            fireTransition(t2)
47:        end if
48:    end for
49: end procedure
```

Algorithm 3 Handles tokens entering a place

```
1: procedure enterPlace
2:     TK = {tk_1, tk_2, ..., tk_n}        ▷ TK is a list of tokens being added to the place
3:     for oe ∈ E_O(p) do
4:         processEvent(oe, TK)
5:     end for
6:     for sm ∈ SM(p) do
7:         beginSubMission(sm, TK)
8:     end for
9:     if p ∈ P_E then
10:        finishPlan(p, TK)
11:    end if
12: end procedure
```

Part II

Monitoring and Interrupting Team Plan

5

A Mechanism to Smoothly Interrupt Team Plan

Team oriented plans are a key tool for allowing human operators to specify high level directives for teams of autonomous agents. In many scenarios an operator might need to interrupt the activities of individual team members to deal with particular situations (i.e., a danger that the team can not perceive). However the way that the system respond to an external interruption is very sensitive to the context of the plan. Our goal is to present a mechanism that allows a range of interrupts to be handled smoothly, allowing the team to efficiently continue with its tasks after an operator intervention.

In this chapter, we describe the basic idea of interrupt in Petri Nets. Then, we provide the details of our proposed interrupt mechanism in SAMI framework (SPNs) following the syntax presented in Chapter 4. Afterwards, we discuss an exemplar multi-robot plan, Cooperative Location Visits, that makes use of such interrupt mechanism[1].

5.1 Modeling Interrupts in PN

Petri Net paradigm does not offer a special construct to implement interrupts, but it is possible to replicate the behavior of an interrupt through a specific sequence of places and transitions [23].

Figure 5.1 reports an example of an interrupt realized in the Petri Net framework. Essentially, the normal execution flow can be interrupted when the system is in *state A*. The interrupt can be triggered by the human operator simply placing a token in the *Interrupt Place*. This will enable the *Interrupt Handler* transition, hence changing the execution flow of the plan. If the *Interrupt Handler* transition fires, the system will place a token in the *End Interrupt* place, and, when the execution of such behavior is completed (i.e., when the *Return to State A* transition fires), the system resumes the normal execution by placing a token back to the *State A* place. Notice that during the execution of the interrupt behavior, the transition *End of State A* is not enabled, therefore the flow of execution can not progress to *State B* until the interrupt handler behavior is completed.

[1] This chapter is based on our journal article: *Interacting with Team Oriented Plans in Multi-Robot Systems* [26].

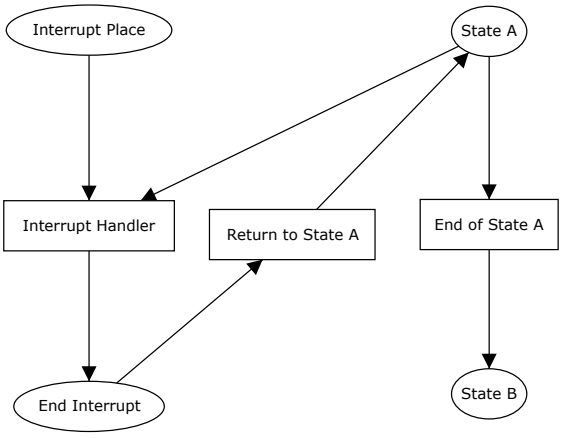


Fig. 5.1. Interrupt implementation with Petri Net.

5.2 Modeling Interrupt in SAMI Framework

Following the interrupt implementation idea described in Figure 5.1, we use three key elements to model the interrupt mechanism in the *SAMI* framework: i) a place (called *Interrupt place*) ii) a transition that starts the interrupt handling procedure (*Start interrupt transition*) and, iii) a transition that determines the end of the interrupt procedure (*End interrupt transition*). Now, consider a generic plan that we represent with a *Source place*, indicating the state of the system that could receive an interrupt, a transition, indicating some part of a plan, and a *Destination place*, indicating the state of the system that should be reached when the interrupt handling procedures terminates (notice that the source and destination places could be the same).

Figures 5.2(a) and 5.2(b) show the CPN structures we propose to add interrupts to. We consider two types of interrupts: a *proxy* interrupt, which concerns the individual(s) and only affects their behavior (see Figure 5.2(a)) and a *general* interrupt that deal with the entire team's behavior (see Figure 5.2(b)). As the figures show, the structure to realize these two types of interrupts is the same; however, the events attached to the places/transitions and the requirements on the edges of the net are different. In both structures, the *Start interrupt transition* and the *End interrupt transition* are connected by a *Sub-mission interrupt place* which represents a sub-mission that models the appropriate interrupt handling behavior. After the execution of the sub-mission all the tokens returned by the sub-mission (i.e., the tokens which completed the sub-mission) move to the destination place of the interrupt, and restore the normal behavior of the plan. Below we describe these two interrupt types in more detail.

Proxy Interrupt The *proxy* interrupt relates to a specific subset of the platforms, and affects the execution flow of those platforms only (while the others continue the normal execution of the plan). This type of interrupt typically represents a procedure that should be activated in response to some proxy-level events, e.g., the battery of a boat reaches a critical level and the boat should stop the current plan to go to a recharge area.

In particular, the interrupt place generates a *Proxy Interrupt*, which is an output event[2]. The *Proxy Interrupt Received* input event encapsulates the information regarding which proxies should be involved in the event. Such information is used by the *Start interrupt transition* to take only the relevant tokens from the *Source place* and move them to the *Sub-mission interrupt place*. Consequently, only the tokens specified by *Proxy Interrupt Received* will stop their current plan to execute the interrupt sub-mission. Such relevant tokens are selected with a plan specific procedure, and this often requires a user interaction (i.e., the user directly selects which platforms should execute the interrupt sub-mission).

General Interrupt The *general* interrupt is a team-level interrupt that is not specific to a particular platform. The *general* interrupt represents a situation where all robotic-boats should perform a particular procedure, e.g., stop all current plans and go to a safe position as a manned boat is approaching.

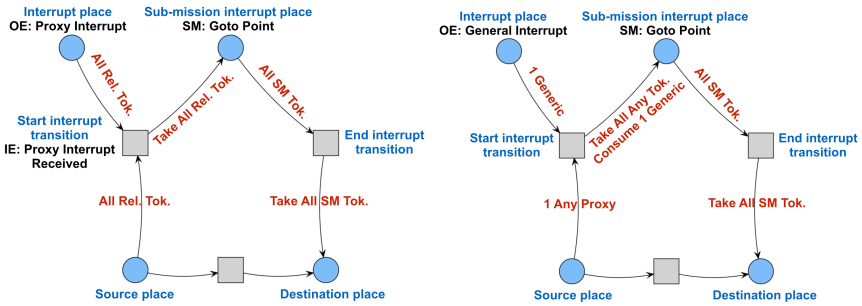
In contrast to the *proxy* interrupt, the *general* interrupt will remove all tokens present in the *Source place* and transfer them to the sub-mission. Hence, the event generated by the *Interrupt place* is a different output event, named *General Interrupt*. Such event is generated to trigger the interrupt mechanism but does not contain any specific information regarding the relevant proxies (as all proxies are relevant in this case). Consequently, the *Start interrupt transition* requires a *generic* token (and not a proxy token) and it will transfer all the *proxy* tokens from the *Source place* to the *Sub-misison interrupt place*. Note that, unlike a proxy interrupt, a general interrupt has no input event on the start interrupt transition, as it always moves all tokens and thus does not require any additional information. A general interrupt is essentially a compact way of representing an interrupt for all proxies. Such compact representation is crucial for team level plans that must be designed and monitored by human operators.

The interrupt parts of the SPN are not logically different from non-interrupt parts. Hence, since SPN supports sub-missions, we can also have nested interrupts. In other words, in both interrupt structures, the *Start interrupt transition* and the *End interrupt transition* are connected by a *Sub-mission interrupt place* which can be a series of *Sub-misisons* that should be executes to implement the appropriate interrupt handling behavior.

5.3 Using the Interrupt Mechanism

Here we provide an exemplar multi-agent plan, discussing the possible use of both interrupt types described above. In particular we consider a Cooperative Location Visit (CLV) plan where the operator selects a group of boats to visit a set of locations to perform point measuring tasks. The boats should navigate to each location and acquire a specific measure (e.g., pH level, oxygen level, temperature). In this work, we assume that each boat is equipped with the same sensors, hence visiting the same location with different boats does not provide more information

[2] Recall from Chapter 4 that output events are associated to places and contain commands or requests for other modules. Input events are associated to transitions and encapsulate information that should be consumed by the module that receives such event



(a) Proxy Interrupt (b) General Interrupt

Fig. 5.2. Types of interrupt implemented in the SPN framework.

and should be avoided, in contrast, each boat can visit several locations (i.e., executing a path that goes through all such locations in sequence). The system offers various techniques to assign boats to locations and in this work we used a method which is based on Sequential Single Item auctions [72]. The method assigns locations to boats sequentially, and for each location the system selects the boat that can provide the lowest path cost. Such path cost is computed as the minimum path cost that the boat can achieve when inserting the current location in the set of locations that are already assigned to such boat[3].

The CLV plan is reported in Figure 5.3. In such a plan, the general interrupt handles a situation where the user decides to temporarily stop the current plan of all the boats to avoid a dangerous situation, i.e., a manned boat that enters the area where the boats are operating. The general interrupt starts from the *Proxy Execute Path* place and goes back to the same place. When the interrupt triggers, all the tokens present in the *Proxy Execute Path* place are transferred to the sub-mission place. This token transfer requires the presence of at least one *Proxy* token in the *Proxy Execute Path* place and is performed by using the *take* action (see Chapter 4) on all Proxy tokens that are present in such place. As mentioned in Chapter 4 the *take* action will remove the specified tokens from the incoming place and will add them to the outgoing place, which in this case is the *Assemble* sub-mission (SPN not shown). Hence the effect of this token transfer is that all proxies will stop executing the current action and will start the *Assemble* sub-mission. Such sub-mission, sends all the boats to a specific safe assemble position and then waits for operator input to end the plan, allowing the parent plan to continue. When the operator decides that the dangerous situation is over, the *End general interrupt* transition fires and boats are sent back to the *Proxy Execute Path* place, where they resume executing the plan, maintaining their previous location assignments. This token transfer does not require the presence of any token (as it is triggered only by the *End general interrupt* event) and it is performed with the

[3] Since computing the minimum path cost given a sequence of visit locations is in general NP-Hard here we use a simple nearest neighbor heuristic: the path is built incrementally by always selecting the next location as the one that is closest to the current location. At the beginning the current location is the boat position.

take action on all sub-mission token. The sub-mission tokens are the set of tokens which reached the end place in the sub-mission; in this case, these are the proxy tokens for the boats which were station keeping to avoid the danger. The *take* action means that the proxy tokens will be removed from the *Start sub-mission* place and added to the *Proxy Execute Path* place.

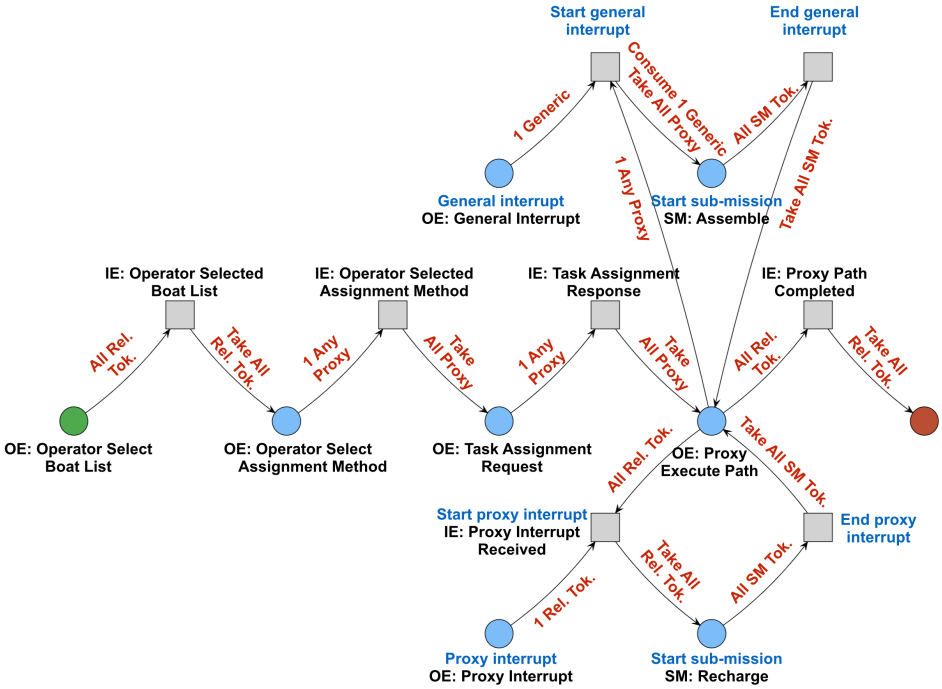


Fig. 5.3. The Cooperative Location Visit plan specified in the SPN framework, with both general and proxy interrupts.

In contrast, the proxy interrupt allows the operator to stop the execution of a selected subset of the boats without interfering with the plan execution of the other boats. This is useful when the human operator should handle an event that influences the behavior of a specific group of boats, i.e., a boat that reaches a critically low battery level. The proxy interrupt moves the set of selected proxies to the sub-mission place while the others will continue their execution. In our exemplar plan, the sub-mission associated with the interrupt, *Recharge*, pauses the current plans of the provided proxies and sends them to a recharge station, where batteries are replaced with fully charged ones. The sub-mission then ends, allowing *End proxy interrupt* to fire, which moves the proxies back to *Proxy Execute Path* where they resume visiting locations. Similar to the general interrupt, we use the *take* action to transfer tokens from the *Proxy Execute Path* place to the *Recharge* sub-mission and then the *take* action to transfer them back. However, in this case we take from the *Proxy Execute Path* place only the *Relevant* tokens, i.e. the

tokens associated to proxies that must be recharged. As mentioned in Section 5.2, the information regarding which tokens are relevant is specified by the input event *Proxy Interrupt Received* associated to the *Start Proxy Interrupt* transition.

Depending on the specific plan and on the desired behavior for the interrupt sub-mission, we might need to insert extra elements into the basic plan. An example of this is the plan to handle the traverse dangerous area event, shown in Figure 5.4 and discussed in detail in Section 5.4.

By combining the team-level and proxy-level interrupts our approach provides a powerful and general model to allow sophisticated interactions between the human operators and the robotic system. As the empirical evaluation shows, this results in a significant performance gain for the system.

5.4 Empirical Results

In this section we present a quantitative evaluation of our approach to team plan monitoring in the water monitoring domain. We first describe our empirical methodology, then we present and discuss the results we obtained.

5.4.1 Empirical Methodology

The main goals of the empirical evaluation are: i) to validate the applicability of the interrupt mechanism to team-level plans that represent realistic use cases, ii) to evaluate the gain achieved by such a mechanism, in terms of task specific performance as well as operator load, with respect to aborting the plan when an incident arises.

We provide a quantitative evaluation of our interrupt mechanism by simulating the plan execution with and without the interrupts in a set of selected use cases. The possibility of repetition of the experiments in the simulation allows us to better evaluate the performance of the system considering statistical significance in the results. Moreover, we validate our approach on real platforms performing various experiments where a human operator should monitor and control the evaluation of several boats.

As a first step, we consider two versions of the CLV plan discussed in Section 5.3: the "interrupt" version which encodes interrupts within the plan (reported in Figure 5.3) and the "standard" version without any interrupts (reported in Figure 4.2). Next, we define three possible incidents: i) *general alarm*, ii) *temporary boat pull-out* and iii) *traverse a dangerous area*. We then simulate the execution of both versions of the CLV plan for each incident, measuring indicators of task specific performance and operator work load. When we execute the standard plan and one of the incidents takes place, the human operator must abort the entire plan's execution, execute the plan that can resolve the incident, and then start a new instance the original plan once the resolution plan has finished.

In more detail, the incidents and the co-related team behaviors have been defined as follows:

General alarm represents a danger that may significantly interfere with the plan execution of all the boats. An example of this could be a manned boat that enters the operative areas of the robotic boats. If this happens the human operator should signal to all the platforms that all plans should be suspended to avoid collisions. When the manned boat leaves the scene the human operator can then instruct the boats to recover the execution of their plans (i.e., execute the remaining tasks). This situation can be handled with a general interrupt as all the boats will have to execute the same specific sub-mission (i.e., reach a safe position) before recovering their plans. In our empirical evaluation we simulate the occurrences of several general alarm incidents while a CLV plan is running. In particular, we fix the number of incidents to happen and distribute them randomly during the plan execution.

Temporary boat pull out represents an incident that interferes with a specific subset of robotic platforms and that will not directly hinder the plan execution for the rest of the team. An example of this could be the need to recharge the battery for one robotic boat. Specifically, we simulate a discharge process for the boats, where the battery level is reduced based on distance traveled. The discharge process includes a random element that increases or decreases the units of battery consumed to simulate possible not-modeled situations (such as currents) that impact the amount of energy required to traverse a given distance. In more detail, if we indicate with $b_i(t)$ the level of battery at time t for boat i, we have that $b_i(t+\tau) = b_i(t) - Kd_i(\tau)(1+R)$, where τ is a positive value that represents a time interval, $d_i(\tau)$ represents the distance (in meters) traveled by boat i in the time interval τ, K is a constant that expresses the units of battery required to travel one meter, and $R \sim U(-0.1, 0.1)$ is a random variable drawn from a uniform probability distribution.

Traverse dangerous area represents an incident where several boats must traverse an area that is problematic for navigation. For example, consider a scenario where a part of the intervention area is cluttered with objects (e.g., vegetation, pieces of wood, etc.) or presents strong currents. In this situation, we require a human operator to constantly monitor the operation of the platforms to be able to promptly intervene (i.e., teleoperating the boats) if necessary. Since it is impossible for a single operator to effectively monitor and teleoperate multiple boats at the same time, a key element for this plan is to synchronize the execution of the boats making sure that only one boat is actively navigating in the dangerous area, while other boats that might need to traverse the same area will wait for the availability of the human operator.

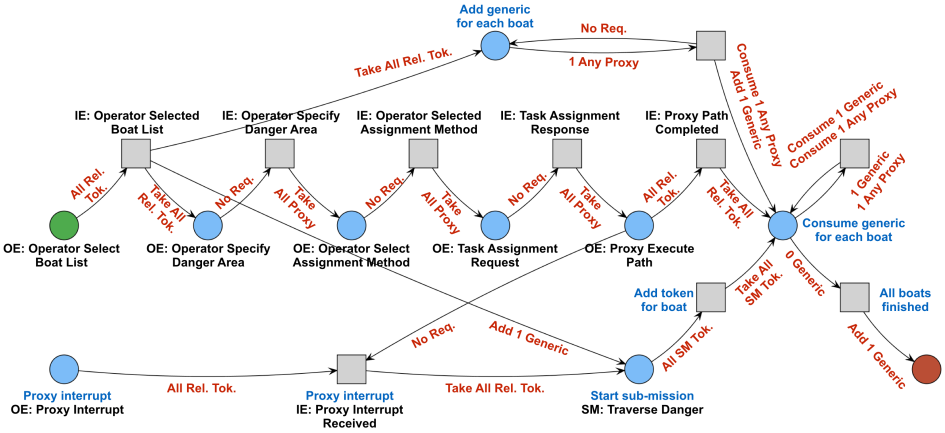
In the standard plan without interrupts, the operator should abort the plan, which means all boats should stop what they were doing. The human operator can then monitor the boats inside the area sequentially. Boats outside the area will be stopped until there is only one boat inside the area, then the plan will resume which means that all remaining tasks will be reassigned. If we execute the plan with the interrupt mechanism, the operator can choose to monitor one platform while all other boats that are inside the area will be stopped until the human operator becomes available for close monitoring. Meanwhile other boats outside the area will continue their paths.

Figure 5.4 reports the CLV plan with a proxy interrupt to handle the traverse dangerous area incident. Specifically, we report the parent plan in Figure 5.4(a) and the traverse dangerous area sub-mission plan in Figure 5.4(b). In the parent plan (Figure 5.4(a)) proxy tokens can follow two different branches to reach the end place of the plan, depending on whether they enter a dangerous area or not. Since in this case the plan should terminate only when all boats have finished their paths (i.e., boats that never entered the dangerous area in addition to boats that did), as mentioned in Section 5.3 we must insert extra transitions and places to make sure that the plan will terminate only when all boats have visited their assigned locations. This is the role of the place labeled *Consume generic for each boat*. In more detail, this place will accumulate one generic token for each platform that is selected by the operator (this is done through the loop in the upper part of the plan). Then, when the proxy tokens representing the platforms reach this place, such generic tokens will be consumed (this is done through the loop in the left part of the plan). The plan will then terminate only when all such generic tokens have been removed. This is done through the last transition (*All boats finished*) which effectively represents an inhibitor arc (it will fire when there are no tokens in the preceding place).[4] The structure of the interrupt is the same as the one reported in Figure 5.2(a), i.e., we have a place that enables the interrupt associated to the output event *Proxy Interrupt* and a start transition for the interrupt (associated to the input event *Proxy Interrupt Received*) that moves only relevant proxy tokens (i.e., only boats that are inside the dangerous area) to the interrupt sub-mission.

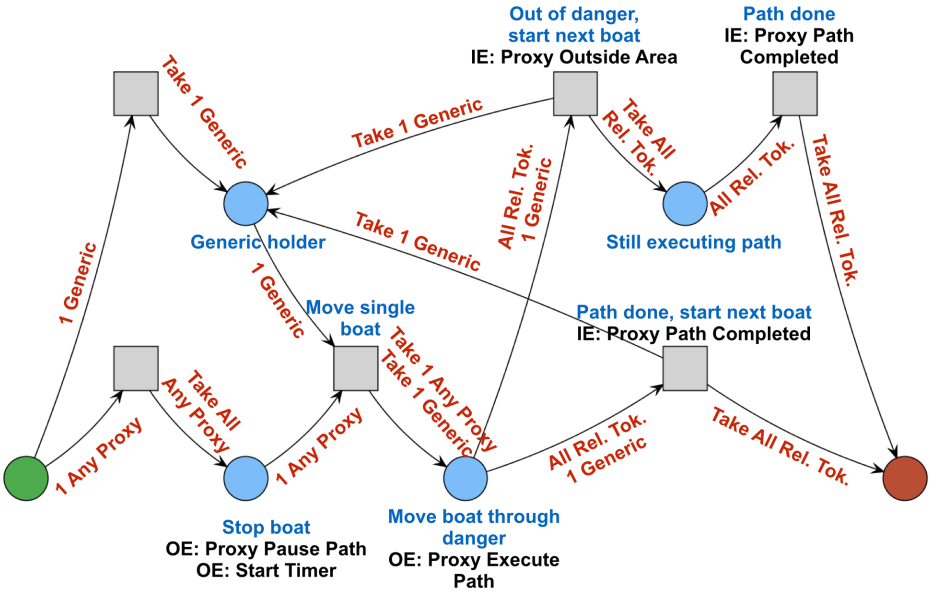
The "Traverse Dangerous Area" sub-mission reported in Figure 5.4(b) is used as static sub-mission (see Chapter 4) in the *Start sub-mission* place. Thus, when the transition holding the input event *Operator Selected Boat List* in the CLV plan fires, the generic token added to the *Start sub-mission* place is also added to the start place of the single instance of the sub-mission. In the sub-mission, the generic token will then be moved to the *Generic holder* place. This place is crucial to synchronize the behaviors of the platforms: if a proxy token enters the sub-mission, the corresponding boat will be stopped and it will not be allowed to execute the remaining path unless there is a token in the *Generic holder* place. Since the transition *Move single boat* takes that generic token, only one boat at a time will be allowed to execute the path inside the dangerous area. The next boat will start the path execution only when the boat currently traversing the dangerous area has completed its path (i.e., when the *Path done, start next boat* transition fires) or it is out of the dangerous area (i.e., when the *Out of danger, start next boat* transition fires). That is because both these transitions put a generic token back in the *Generic Holder* place. Note that, these two transitions are mutually exclusive, so it is not possible for both of them to trigger, which would result in two generic tokens being place in *Generic holder*. Overall, this plan represents a complex team oriented plan that requires a sophisticated synchronization between the boats, however the interrupt mechanism and the use of advanced features of the SPN framework (such as the static sub-mission) allows us to realize such a

[4] While in our case the number of proxy / generic tokens is always finite, we might not know this number before the plan starts. Hence we use the inhibitor arc to check whether a place is empty.

plan in a fairly compact structure.



(a) The parent plan



(b) The (static) sub-mission for the traverse dangerous area

Fig. 5.4. CLV plan with the interrupt for traverse dangerous area

Execution model for the system In our experiments we adopt the following execution model for the system: when we execute the interrupt version of a plan, with interrupt mechanisms in place, we assume that whenever an incident requiring intervention arises, the operator will trigger the corresponding interrupt. For example, when we execute the CLV plan and the battery level of a boat reaches

a critical level, in our simulation the corresponding proxy interrupt will always be triggered and the correct boat will be selected. In other words, we assume the human operator will always do the correct actions that the framework offers to respond to an incident. This is because our intent here is to evaluate the interrupt mechanism and not the human interface. A proper evaluation of the human interface falls outside the scope of this work.

When we execute the standard version of the plan, which lacks interrupts, we assume that the human operator will abort the current plan, start a new plan(s) to handle the incident and, finally, when the incident has been resolved (e.g., a low battery has been swapped), they will start a new instance of the original plan to complete its objectives. When the operator starts the new instance of the original plan, all required information must be re-inserted, such as the locations to visit. In our experiments, we assume the operator can keep track of which locations have been visited and re-start the plan only with the locations yet to be visited (reducing the number of interactions in favor of the standard approach). Moreover, we assume that the operator will start the new instance of the original plan only after the plan(s) used to resolve the incident has been completed. For incidents which do not affect the entire team (e.g., a boat with a low battery requiring a pull out and a subset of the team needing to traverse a dangerous area), this means that some of the team will remain idle when the original plan is aborted, even though they are not involved in the incident.

We further investigate this with a second set of plans for the temporary boat pull out scenario. In these "reassignment strategy" versions of the standard and interrupt plans, when a boat leaves to swap its battery, the rest of the team continues with its tasks. Furthermore, we reassign the locations that boat was responsible for to the other members of the team. When the battery swap is finished, we reassign all tasks that must still be accomplished to all boats. While the commands sent to the boat team are identical for the standard and interrupt versions of the plan for the reassignment strategy, the actual SPNs and the way the operator interacts with them to respond to the low battery incident are different.

Metrics The metrics we extract from the simulation combine task dependent metrics and metrics to evaluate the operator load. Specifically, the task dependent metric is the time to complete a plan while the load metric is the number of user actions required to start/abort the plan, trigger the interrupt, provide information to the boats (e.g., the locations to visit). In our experiments, such interactions always take the form of a click (on a map or on a button), hence, we measure the number of clicks that the operator performs. Since the main goal of the empirical evaluation is to compare the use of the interact mechanism with the standard execution model, we compute and report the percentage gain of the interrupt mechanism for both metrics. In particular, we compute $\frac{(v_{Std}-v_{Int})}{\max\{v_{Int},v_{Std}\}} * 100$, where v_{Std} is the value of the metric obtained with the standard execution model and v_{Int} is the value of the metric obtained with the interrupt mechanism. Since for both metrics the lower the better, a positive value indicates superior performance of the interrupt mechanism over the standard execution model.

In all the following experiments, the interrupt mechanism does not provide additional domain knowledge with respect to the standard plan execution. In particular, the recovery procedure for handling the incidents is the same when using

interrupt and when aborting plans. Overall, our goal here is to provide a domain-independent interrupt mechanism which can be applied to a variety of different incidents such as dangerous area, by raising a domain-specific recovery function. Moreover, we aim at doing this in a smooth way (i.e., without stopping and restarting the plan that is currently running). While one could potentially devise a different domain-specific mechanism to select the most suitable recovery procedure this would defeat the purpose of using a general plan specification language such as SPN.

In this perspective, the gain we obtain is due to the presence of the interrupt mechanism that smoothly changes plan execution instead of aborting and restarting. Consequently, in most situations the interrupt mechanism will require fewer interactions, because we need at least the same number of user interactions to stop and re-start the plan compared to interrupting it. However, for completion time there might be situations where having the interrupt mechanism does not help (e.g., see results for Table 5.1).

In the next section we report and discuss the results obtained with our empirical evaluation.

5.4.2 Quantitative Results in Simulation

Table 5.1 reports the results obtained for the CLV plan and the boat pull out incident. In particular, we consider a set of configurations, where each configuration is defined by three elements: i) the number of boats involved in the plan (3,5), ii) the number of locations to be visited (20,30) and iii) the time required to exchange a boat's battery expressed in seconds (10,20). For each configuration we executed 10 repetitions. We report the average values of the gain for both metrics and the standard error of the mean (shown in square brackets). In the tables, we report only the percentage gain for configurations that show a statistically significant difference between the values of the means[5].

Configurations	Std	Int.	% Gain (Interrupt vs Standard)	
#boat,#loc.,r.t.	#rec.	#rec.	Total Time	# interactions
3, 20, 10	6	6	6.3%	73%
5, 20, 10	5	5	23% [± 0.5]	68%
3, 20, 20	6	6	26% [± 2.5]	72% [± 0.8]
5, 20, 20	5	5	27% [± 6.6]	64% [± 3.7]
3, 30, 10	11	12	26% [± 1.2]	69% [± 9.5]
5, 30, 10	10	12	21%	75%
3, 30, 20	11	12	48% [± 0.8]	80% [± 0.1]
5, 30, 20	10	12	27% [± 2.9]	75% [± 0.5]

Table 5.1. Results for the CLV plan and boat pull out event. Each configuration specifies the number of boats, the number of locations, the time required to recharge the boat's battery (in seconds). The number of recharge (#rec) represents the number of times a boat required a recharge action for the standard execution (Std.) and for the plan with the interrupt (Int.)

[5] To check whether results are statistically significant we run a t-test with $\alpha = 0.05$.

As it is possible to see, for all configurations the plan with the interrupts achieves better performance both in terms of time to complete the plan as well as for the operator workload. In more detail, focusing on the time to complete the plan, we can see that the gain of the interrupt mechanism with respect to the standard mechanism increases when the recharge time increases, because in the standard execution model all plans must be aborted when a boat must recharge, while in the interrupt model the other boats can continue with their plan execution. As for the operator work load, the interrupt mechanism requires far fewer user actions than the standard plan. This is due to the fact that, in the standard execution model, the user must re-insert the locations that the boats must visit when the CLV plan is re-started.

The number of recharge actions is higher when using the interrupts model. This is because the standard mechanism re-starts the whole plan each time a boat must be re-charged, consequently the remaining locations to be visited will be re-allocated among the currently available platforms. This provides solutions of higher quality for the allocation process (i.e., shorter paths), compared to the interrupt mechanism, which uses the same solution throughout the entire plan execution. Therefore, when using the interrupt mechanism boats might end up traveling more, and since the battery discharge process depends on the traveled distance, this results in more recharge actions. However, as results clearly show, this is compensated by a significant reduction in time to complete the plan and operator load.

Configurations	% Gain (Interrupt vs Standard)
#boat,#loc.,#alarms	# interactions
3, 20, 1	44% [± 0.6]
5, 20, 1	40% [± 1.4]
3, 20, 3	65% [± 0.6]
5, 20, 3	61% [± 1]
3, 30, 1	46% [± 0.3]
5, 30, 1	16% [± 1.9]
3, 30, 3	68% [± 0.23]
5, 30, 3	66% [± 0.4]

Table 5.2. Results for the CLV plan and the general alarm event. Each configuration specifies the number of boats, the number of locations and the number of alarms.

Table 5.2 reports the results achieved for the CLV plan and the general alarm incident. We considered the same number of boats and number of tasks, and we vary the number of alarm incidents that will appear during the plan (1,3). As before, we report the average values of the gain and the standard error of the mean.

Concerning the operator work load, these results confirm the superior performance of the approach that encodes interrupts in the plan. However, in this case, the difference in time to complete the plan does not show a statistical significance, consequently we do not report such values. This is because the procedure to handle the general alarm requires all boats to stop and wait until the original plan can

Configurations	% Gain (Interrupt vs Standard)	
#boat,#loc.#boats inside area	Total Time	# interactions
3, 20, 2	5.2% [± 2.9]	40.2% [± 2.16]
5, 20, 2	6.9% [± 2.2]	39.1% [± 0.5]
3, 20, 3	10.4% [± 1.7]	42.5% [± 0.6]
5, 20, 3	9.8% [± 1.8]	42.9% [± 1.1]
3, 30, 2	(4.3% [± 2])	45.3% [± 1.6]
5, 30, 2	9.9% [± 2.4]	43.6% [± 1.3]
3, 30, 3	5.4% [± 1.3]	43.6% [± 0.5]
5, 30, 3	15.9% [± 1.7]	44.4% [± 0.5]

Table 5.3. Results for the CLV plan and enter dangerous area event. Each configuration specifies the number of boats, the number of locations and the number of boats that are inside the dangerous area at the same time (the value between parenthesis is not statistically significant according to a t-test with $\alpha = 0.05$, all others are).

	Simple Strategy		Reassignment Strategy
Configurations	% Gain (Interrupt vs Standard)		% Gain (Interrupt vs Standard)
#boat,#loc.,#rec,r.t.	Total Time	#interactions	# interactions
3, 20, 3,10	11%	65%	80%
5, 20, 3,10	16%	65.4%	81%
3, 20, 3, 20	14.8%	64%	79.6%
5, 20, 3, 20	13.4%	63.4%	78.7%
3, 30, 5,10	13%	75.6%	86%
5, 30, 5,10	17%	73%	85%
3, 30, 5, 20	16.8%	76%	86%
5, 30, 5, 20	11%	76.6%	83%

Table 5.4. Results for the CLV plan and boat pull out incident for the previous simple strategy (do not reassign tasks) and reassignment strategy. Each configuration specifies the number of boats, the number of locations, the time required to recharge the boat's battery (in seconds). The number of recharge (#rec) represents the number of times a boat required a recharge action which is assumed to be 3 for 20 locations and 5 for 30 locations in these experiments.

be safely re-started. Hence, the actions that the boats perform when aborting a plan are very similar to the interrupt handling procedure. Notice that, in all the simulations we do not consider the time required by a human operator to perform the click actions but we simply count the number of clicks. This is because a proper evaluation of such time would be highly dependent on the skills of the operator. However, in practice this time will not be negligible and would significantly increase the gain in favor of the interrupt mechanism.

Table 5.3 presents the results for the CLV plan with the traverse dangerous area incident. Again, we consider the same number of boats and tasks and we vary the number of boats that simultaneously enter the dangerous area during the plan (2,3). In this case, if a single boat is inside the dangerous area there is no need for interrupting the plan. This is because the plan monitoring framework allows the operator to override boat autonomy at any time, directly teleoperating a single platform without aborting the current plan. Hence, if a single boat is

traversing the dangerous area the operator can focus his/her attention on such a boat without changing the behaviors of the other platforms. However, if more than one platform are traversing the dangerous area at the same time, the plan must be changed to stop all boats inside the area so to focus operator attention on a single one. Hence, in our experiments, we consider only situations where at least two boats are simultaneously inside the dangerous area.

Results show that also for this type of incident the interrupt mechanism provides an important gain (about 40%) in operator load and that such a gain does not vary significantly across the considered configurations. This is reasonable as the number of interactions that the operator must perform does not depend on number of boats and only marginally on the number of visit locations: in the standard version of the plan the operator will have to re-insert a higher number of locations when re-starting the plan, this is confirmed by a small increase in the gain when there are 30 locations to visit. As for the completion time, the gain is less significant and there is no clear trend with respect to the configurations we considered. In fact, in this case, the gain depends on how tasks are placed with respect to the dangerous area. In any case, the use of our interrupt mechanism is providing a positive gain in all the configurations we considered.

Table 5.4 shows the results obtained for the CLV plan and the boat pull out incident using two different incident handling strategies, as described above. The goal of this set of experiments is to assess the flexibility of our interrupt mechanism and investigate whether the efficiency of the interrupt structure is dependent on the use of particular sub-missions. We consider the set of configurations used in Table 5.1, but to better compare the two plans we now assume a fixed number of recharge incidents during the plan (i.e. 3 boats pull out incidents for 20 locations and 5 incidents for 30 locations). The first two columns present the results using the same handling strategy and plans as in Table 5.1, while the third column shows the results for number of interactions for the reassignment strategy version of the standard and interrupt plans[6]. As mentioned previously, in the reassignment strategy versions of the plans, whenever the boat pull out incident occurs, the related boat will go to the base station for recharging while the remaining tasks are reassigned to the other boats, which continue visiting their assigned locations. When the boat is recharged, all the locations that must still be visited will be reassigned to all boats (including the recharged one).

Results show that the total time gain for the reassignment sub-mission interrupt mechanism according to this metric is not significant. This is expected as in both the standard and interrupt plans, the boats are never idle, unlike the simple strategy version of the standard plan. However, the gain for number of interactions (clicks) significantly increases. This is because, when the interrupt mechanism is not used, the operator needs to reassign the tasks when the recharging boat goes to the base station and when it comes back. In contrast, when the interrupt mechanism is used everything is handled through the sub-mission hence there are fewer interactions. In summary, the key point is that the interrupt mechanism helps in

[6] According to a t-test with $\alpha = 0.05$, the total time gain for the reassignment versions of the interrupt versus standard plan is not statistically significant, so we do not report such metric in the table.

terms of completion time and interactions, and it is a flexible and general approach that can be easily used with different sub-missions.

Finally, a video showing an exemplar execution of the CLV plan presented in Figure 5.3 is reported here[7]. The video shows that, when the general interrupt is triggered all the boats move through the interrupt branch and enter a recovery sub-mission that sends them all to a safe assembly location. When the alarm is over, the boats resume their previous plan. In contrast, when the proxy interrupt is triggered, the selected boat proceeds to the recharge area while the execution of the other boats progresses unchanged. When such boat completes the recharge plan, it returns to finish executing its previous plan.

The video shows how our mechanism allows the human operator to smoothly handle different types of interrupts during the execution phase of complex team-level plans.

5.4.3 Validation on robotic platforms

We validated the use of our approach for interacting with team oriented plans on real robotic platforms. Specifically, we performed several experiments where a single operator was in charge of monitoring and interacting with the operation of several boats (up to nine). Here, we discuss a specific experiment where platforms are sequentially inserted into the water and, as they are added, they start to execute a *Connect and station keep plan* to maintain a specific predefined position. A video of an exemplar run for the connect and station keep experiment can be found here[8] while Figure 5.5 reports a picture of the same run.

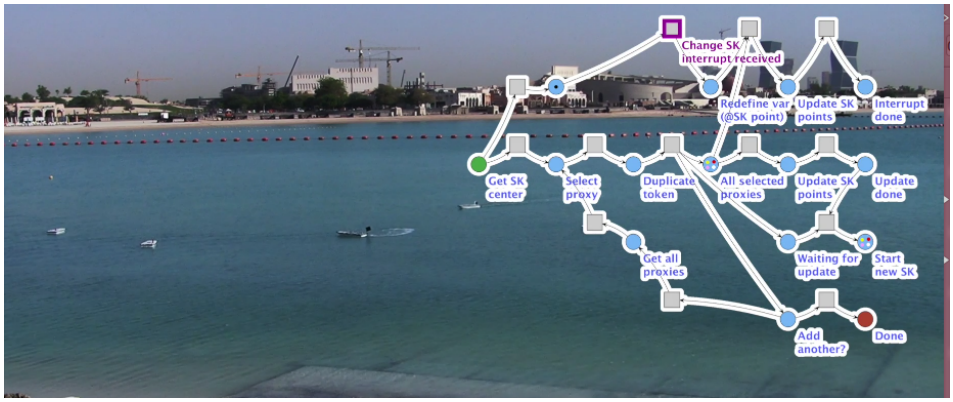


Fig. 5.5. A picture of the connect and station keep experiment. The image shows a subset of the platforms and the current state of the CPN representing the connect and station keep plan. The interrupt portion of the plan is visible in the top part of the picture and the current enabled transition (highlighted in the picture) is the one that starts the interrupt to change the position where boats should perform station keeping.

[7] http://profs.sci.univr.it/~farinelli/videos/CLV.mp4
[8] https://youtu.be/l5Qhp1JSoNI

The experiment has been conducted in a marine coastal area, and as it is possible to see, currents would make the boats float away when motors are shut down. To avoid this, when executing the connect and station keep plan, the boats will periodically turn on their motors to move toward a assembly positions specified when the plan is invoked (left of the screen). This is a crucial behavior to effectively deploy a large team of platforms. The video shows the boats executing the plan, the evolution of the CPN representation for this plan, and a few screen-shots of the graphical interface that the operator uses to monitor the plan.

In this experiment the interrupt mechanism is used to re-define the points where boats should perform station keeping. This is a general interrupt as all boats will change their behavior. The operator activates the interrupt at minute 1:50 of the video, and it is possible to see how all boats change their plan and perform the station keep behavior in a different position (center of the screen).[9] This behavior is used in field deployments when large speed boats approach the current station keeping location, risking a collision with the robots.

These experiments confirmed that our interrupt mechanism helps human operators to easily control the deployment of real robotic platforms.

5.5 Summary

In this chapter we discussed the motivation, implementation, simulated evaluation, and the field operation of the SPN interrupt mechanism which allows an operator to quickly trigger complex behavior. Interrupts were a key feature of the SPN language in field deployments and nearly all plans in CRW framework have at least 2 interrupts. Support for globally scoped variables also proved to be important, as it allowed for information to be shared across plans and their interrupt behaviors. For instance, adjusting the safe recovery location in one plan due to receded tides or a newly docked boat could then be carried across to other plans.

In the next chapter, we will focus on decision making for specific interactions in multi-robot applications, where each robotic platform decides whether to ask for the operator's help or not. Though we use the same application domain, i.e. robotic water monitoring application, team plan representation is not our concern. In other words, the proposed mechanism is more general that can cover a wide range of applications with similar structures, where a team of multiple robots are controlled by a single operator and the robots can initiate requests for human interventions.

[9] This video was accepted to the IJCAI 2015 video competition.

Part III

Self-Reflection and Autonomy in Human-Multi-Robot Interactions

6

Investigating Balking Strategies in Cooperative Multi-Robot Systems

The supervisory role of the human operator can become critical, when the number of robots demanding the operator's attention increases. The operator cannot handle all requests simultaneously, hence, the robots should wait for the operator. Queuing is a natural way to manage and address the requests sequentially. Previous research try to enhance the performance of the system (i.e decreasing the time spent by robots waiting for the opearor) considering the queue disciplines (e.g. FIFO, SJF, etc.) or prioritizing the requests. In contrast, we focus on designing queue structures, where the robots decide whether to wait for the operator or not. To do this, we assume that the robots are able to identify their requests and can decide autonomously. In more detail, we consider the *balking queue* [46] structure and adjust its parameters to make it applicable in a robotic application. In particular, we discuss the idea of balking queue strategies, where the robots decide whether to join the queue or not following the general model presented in Section 3.4. Then, we provide the details of our proposed formalisms to model human-multi-robot interaction as a balking queue. More specifically, we are looking at threshold strategies, which are based on dynamic features of our robotic water monitoring environment and use Reinforcement Learning to compute these cooperative policies. Finally, we validate our balking models in an exemplar human-multi-robot scenario [1].

6.1 Problem Definition

Adding self-reflection to robots (e.g. by warning the operator or asking his/her permission) may help the operator to be more focused on his/her supervisory role. However, keeping robots idle until the operator becomes available might decrease the overall team efficiency. For example, using a FIFO or SJF or any other types of queue without balking forces a robot with a request to wait in the queue [20,42]. Consequently, a significant amount of time will be spent waiting for the operator's response which would affect the team performance when the time is a critical asset.

[1] This chapter is mainly based on our conference paper: *A Balking Queue Approach for Modeling Human-Multi-Robot Interaction for Water Monitoring* [56].

In our robotic water monitoring application, there is a single operator controlling the team of robotic boats. We assume that a set of specific events, see table 6.1, can happen to the platforms. Such events may affect the normal behavior of the platforms and hinder their performance.

Following previous works [20, 42], we consider a central queue is provided to both the operator and the boats, where the operator can select one request at a time and assign a specific sub-mission to resolve that request. These sub-missions range from giving permission to a boat entering a specific area to ones that need teleoperating a boat. Recall from Chapter 5 that, a sub-mission is a plan specific recovery procedure, and this often requires a human interaction (i.e., the human directly selects which platforms should execute the interrupt sub-mission). For example, in our experiments we used three different sub-missions, one for each class of requests as following:

- **Recharge** send a boat to the closest station to change/charge its battery.
- **Permission** allow/not-allow a boat to go further (to the area that it might loose connection).
- **Teleoperation** give control to the operator to teleoperate the boat traversing a specific area.

We assume that, whenever an event happens, the platform can detect the event. For example, the robot can perceive that its battery level is in a critical state, then it can send a request for the operator's intervention. Even though, this ability is limited and may not be possible in some domains, this is not the focus of this thesis. The platform must then decide whether to join the queue (i.e. sending the request and waiting for the operator) or balk (i.e. not sending the request). We consider a FIFO queue for arrival requests, where the operator can only select and resolve one request at a time.

Following the proposed model in [46], we map the human-multi-robot interaction part of our application into a balking queue structure, in which the boats can choose to join the queue or balk according to a threshold value. In our scenario, each boat will increase the team waiting time by joining the queue and may end up to a failure by balking the queue. Each event type has a different severity according to table 6.1, where the higher severity requests are more crucial to receive the operator's attention. We assign a probability of failure to each event type according to its severity (i.e. the higher the severity, the more probability of failure). We use these probabilities as quantitative inputs for the simulation. Notice that, the balking consequences or costs are problem specific. In our model, when a failure happens, the operator should spend more time to fix the problem. So, failure as a consequence of balking, brings extra cost to the system.

As an example, consider event E_j (see table 6.1) happens to boat i, and it must select to join or balk. The decision of joining the queue will affect the future decisions of other boats, while choosing to balk with some probability (regarding to event type) may result in failure. The goal is to minimize the time spent in the queue while keeping the number of failures low. However, defining appropriate behavior for each robot is not trivial since, the robots do not know about the future events/requests that may occur (i.e. to the platform itself or other team

Event Type (E_j)	Severity	Probability of Fail
Battery Recharge (E_1)	High	0.9
Traversing Dangerous Area (E_2)	Med	0.4
Risk of Loosing Connection (E_3)	Low	0.2

Table 6.1. Different event types used in the experiments. We assign a probability of failure to each event type according to its severity.

members) in the system. Our aim is investigating how the elements of balking strategy should be computed according to this practical robotic scenario

In what follows, we present three main approaches for computing balking policies in the above problem. First, in Section 6.2.1, we introduce the dynamic balking threshold instead of using static threshold value as it is usually done in the literature [46]. Next, in Section 6.2.2, we learn the balking policies for a single robotic boat, while the others using the dynamic threshold. Finally, in Section 6.3, we consider multi-agent reinforcement learning (MARL) approach for computing the balking strategies.

6.2 Single-Robot Balking Policies

6.2.1 Dynamic Threshold

In Section 3.4, we explained the general balking model [46], where a static cost C for staying in the queue and a static reward R for receiving service are assigned to all requests or users (see Equation 3.6). However, in our dynamic threshold model, instead of those fixed static cost and reward values, we consider the dynamic features of our robotic application to design different reward values for receiving a service and different cost values for waiting for the operator.

In more details, in our water monitoring application, there are different types of requests, each with a different severity (see Table 6.1). Requests with higher severity are more critical to receive the operator's response, because there is a higher probability of failure when balking these kinds of requests. Hence, we consider the severity of a request type as an important feature in the reward function associated to that request. That is, a higher severity request will obtain a higher reward value if it receives a response from the operator (e.g. by joining the queue).

Another effective feature, specific for this domain, is the number of unfinished tasks (or unvisited locations) of a boat at the time of sending a request. For example, the cost of waiting in the queue for a boat with only one unvisited location is lower than the waiting cost of a boat with several unvisited locations. Thus, the number of unvisited locations is considered in the waiting cost function of each request. Notice that, this value depends on the number of unfinished tasks of a boat at the time of sending a request (i.e. it varies during the mission execution).

To sum up, the balking threshold for $boat_i$ with k unfinished tasks at time t with a request type $type_j$ is the following:

$$n_{threshold} \leq \frac{R(type_j)\mu}{C(k)}. \tag{6.1}$$

where $R(type_j) = \alpha \times (ProbFail(type_j))$ is the reward function associated with the request $type_j$ and $ProbFail(type_j)$ is the probability of failure of request $type_j$. $C(k) = \frac{\beta}{k}$ is the cost function of $boat_i$ with k unfinished tasks [2]. Equation (6.1) indicates that, $boat_i$ at time t joins the queue if and only if the number of requests inside the queue is not more than $n_{threshold}$. As you see, $n_{threshold}$ is a dynamic value, including the state of the queue and the state of a boat (i.e. type of the request and number of unvisited locations). Two functions $R(type_j)$ and $C(k)$ are adjusted based on the designer experience considering the average arrival rate, average service rate and the probability of failures.

Even though, we show the benefit of using this model through experiments (see Section 6.2.3), however, this model does not consider a sequential decision process but only computes one-step decisions.

Our proposal is then to train the robots in a stationary environment (i.e. stationary distribution functions with fixed arrival rate and service time), so that the robots can learn appropriate balking policies. Then, by applying the learned policies in the same environment, they will be able to optimize the team objective.

6.2.2 Single-Robot Learning

In this section, we apply Q-Learning to find the optimal balking strategies in our water monitoring application. Because of its simplicity, Q-learning is a fair choice where we have access to a simulator (i.e. no limitation on generating samples), the state and action spaces are discrete and keeping the Q-values table in the memory is practicable.

This application scenario has specific attributes, including homogeneous robots with highly independent tasks. Considering these properties, we can use the single-robot setting, in which one robot learns a balking policy, while the others use the threshold values in equation (6.1). Then, during the experiments, the learned values (i.e. Q-values learned by one robot) will be utilized by all team members.

In Q-learning, considering a single-robot, the robot will select its action according to a potential stochastic policy and will update its policy by greedily maximizing the Q-values. The Q-value at each time-step, will be updated according to Equation (3.5). More details of Q-learning are given in Chapter 3.

According to our application, action and state spaces are defined as follows:

- The action space A includes $\langle Join, Balk \rangle$;
- The state space S of the learner boat is a tuple $\langle N_q, N_{tasks}, S_b \rangle$ where:
  - N_q represents the number of requests inside the Queue
  - N_{tasks} shows the number of remaining tasks of the boat
  - S_b is the current internal state of the boat. For example, whether it has a request, if it is waiting for the operator (in the queue), etc.. More specifically $S_b \in \{E_j, \mathbf{W}aiting, \mathbf{F}ailed, \mathbf{A}utonomy\}$ where $j = 1, 2, ..., n$ is the cardinality of the event types. In our model, E_j refers to one of the types in table 6.1 where $n = 3$. As an example, the state tuple of a boat when

[2] α and β are tuned empirically.

the current length of the queue is 2, it has 3 tasks to finish and it comes up with a request type of *Battery Recharge*, would be $s = \langle 2, 3, E_1 \rangle$

The general rule for immediate reward r in this model is the following: when a boat joins the queue with length N_q, it receives $\frac{1}{N_q}$ reward. When it balks, it may fail where in this case it receives $R_F = -2$ reward. However, if it does not fail, a reward $R_T = 0.3$ would be assigned to the boat. We use ϵ_greedy method with parameter $\epsilon = 0.1$ in action selections. Our algorithm uses the learning rate $\alpha = 0.1$ and discount factor $\gamma = 0.9$ throughout the experiments.

All the above elements such as ϵ, α, γ and the reward values have been tuned empirically.

As mentioned before, the policies learned by this boat will be used by other boats as well. In the next section, we will evaluate the performance of these two models in our multi-robot simulation scenario. In the Section 6.3.1, we will explain a more sophisticated formalism to model and solve the problem in a multi-robot learning settings, where all boats learn simultaneously.

6.2.3 Empirical Evaluation

In this section, we evaluate the use of our balking queue model within a simulation of water monitoring application.

Our aim is to decrease both the overall team waiting time and the total number of failures. A large number of failures shows that, the balking decisions of boats are not reasonable even if these decisions decrease the waiting time. In other words, as mentioned before, failures bring extra cost for the operator and the entire team since the operator should spend more time to resolve the problem. For this purpose, we evaluate the above models (Dynamic Threshold and Q-learning approach), under the following setups:

- In all cases, we consider a single operator (server) responding to different requests from boats.
- The mission of the team is generated by the operator, where the operator assigns a list of locations to be visited to each boat. Five boats and thirty tasks are considered for each mission and all experiments.
- For our experiments, three types of request, each with different severity has been considered as table 6.1 shows. The service rate and arrival rate of each kind of request are assumed to be independent and exponentially distributed.
- A mission finishes after the occurrence of a fixed number of events (i.e 30 events).

In the first set of experiments, we programmed all the boats to follow the dynamic threshold computed in Equation (6.1). For each set of configurations, we run 20 trials, and we report the average over all such runs. Both the service times and arrival times are independent and exponentially distributed with rate parameter μ and λ respectively. We used a realistic estimation for parameter λ and μ based on some experience on the total mission time, number of boats and number of locations. In more details, events happening with the rate $\lambda = 0.25$, where the time between events (inter-arrival time) has a mean of $\frac{1}{\lambda} = 4$ seconds

and is modeled using an exponential probability distribution. Service time for each request is also generated with a exponential probability distribution with mean value $\mu = 0.27$. These numbers provide a good trade-off between boats that can operate in autonomy but requires intervention. For example, a very big arrival rate shows that, the boats are not autonomous at all (i.e. they cannot perform their tasks without the operator), while a very small arrival rate shows that, the boats are in a very high level of autonomy. As mentioned before, our focus is on situations, where the boats are neither fully autonomous nor fully dependent to the operator.

We aim to compare the behavior of the proposed model to FIFO and SJF without balking strategy.

Table 6.2 shows the results for 5 boats and 30 tasks. As the results show, the waiting times in the queue for the dynamic threshold approach are less than FIFO and SJF model, since in the former, not all requests join the queue. Notice that, decreasing the waiting time in the queue, will reduce the total time for a mission as well. In this table, because SJF and FIFO models do not balk any requests, hence the number of failures will be zero. In our model, failures only happen for balking. This assumption is in favor of non-balking models. For example, if a boat waits too long for the operator the battery might run out, thus the mission fails just because time passes. Hence, in practice, the results will probably be even more in favor of our approach.

Table 6.2. Results for 5 boats, 30 tasks, $\lambda = 0.25$ and $\mu = 0.27$. Each column shows the average value over 20 simulation runs. All times are in seconds. Results are statistically significant according to a t-test with alpha = 0.05.

Queue Model	#Req	Total w.t.	%Balking	%Failure
Dynamic Thresh.	30	139	34%	14%
SJF	30	256	0	0
FIFO	30	356	0	0

In the second set of experiments, we used Q-Learning approach for one boat to learn the threshold policies, while the others follow the dynamic threshold policy. For training one boat, we use the same configuration (e.g. same λ and μ) as used for the dynamic threshold. Each episode (i.e. a run of the algorithm beginning from a start state to a final state [68]) stops, if the learner boat ends up to a failure (because of balking a request) or it finishes all assigned tasks for its mission. Figure 6.1 plots the mean cumulative rewards at each episode of these experiments, where you can see the convergence of our Q-Learning approach.

Finally, we use the same Q-values computed by one boat, for all boats to see how the behavior of the system changes. Table 6.3 shows the result of Q-Learning for one boat (QL-Single Boat) and Q-Learning for all boats (QL-All Boats), where we use the same simulation setup as the first experiment. As the results show, the percentage of balking in QL-All Boats is less than the other methods, thus the waiting time in the queue increases. However, the percentage of failures has fallen substantially in QL-All Boats in comparison to dynamic threshold and QL-

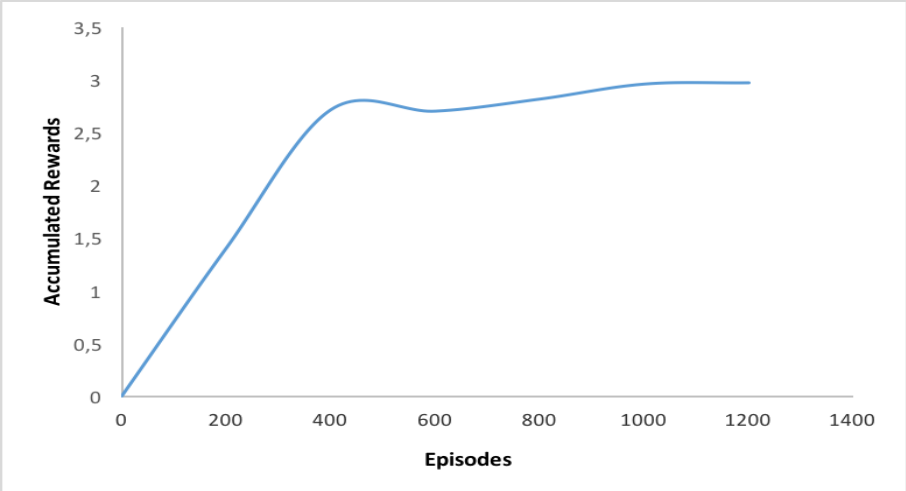


Fig. 6.1. Learning curve of a single robotic boat. The accumulated rewards are averaged over 100 episodes.

Single Boat. Since the waiting time is still better than approaches without balking, depending on how critical is a failure, our results suggest that QL-All Boats might be better than the other approaches.

Table 6.3. Results for 5 boats, 30 tasks, $\lambda = 0.25$ and $\mu = 0.27$. Each column shows the average value over 20 simulation runs. All times are in seconds. Results are statistically significant according to a t-test with alpha = 0.05.

Queue Model	#Req	Total w.t.	%Balking	%Failure
Dynamic Thresh.	30	139	34%	14%
QL-Single Boat	30	163	28%	9%
QL-All Boats	30	209	10%	1%

6.3 Multi-Robot Balking Policies

6.3.1 Model Description

The Q-Learning model used in Section 6.2, finds the local balking policy for one robotic boat, while the other boats using a predefined balking threshold. The main issue with that approach is that, the learned values are biased towards the dynamic threshold (Equation 6.1) of the team. In other words, we assume the other robots have a known behavior, and a single robot tries to learn those behaviors in order to improve the total team rewards. In contrast, in this section, our aim is to make the team of robots learn cooperative balking strategies to make better use of a

shared queue without any prior knowledge about each other's policies. To do that, we propose to cast this problem as a multi-agent reinforcement learning problem.

In particular, we frame the above problem as a Decentralized Markov Decision Process (Dec-MDP) in which the team of agents must cooperate to optimize some global objective (i.e. decreasing the time spent in the queue concerning the probability of failures), while each agent only observes part of the world's state.

we consider the following model:

- The state space $S = S_1 \times S_2 \times ... \times S_N$ where N is the number of boats. The local state of each boat S_i is a tuple $\langle S_b, N_{tasks} \rangle$ where:
  - N_{tasks} shows the number of remaining tasks of boat i. As mentioned before, in this application domain, tasks are a set of locations that should be visited by the boats.
  - S_b is the current internal state of boat i (e.g. whether it has a request (which type), if it is waiting for the operator, etc.). More specifically $S_b \in \{E_j, \mathbf{W}aiting, \mathbf{F}ailed, \mathbf{A}utonomy\}$ where $j = 1, 2, ..., m$ is the cardinality of request/event types. In our model, E_j refers to one of the types in table 6.1 where $m = 3$. For example, the state tuple of a boat when it has 3 tasks to finish and the event *Battery Recharge* occurs, would be $s = \langle E_1, 3 \rangle$
- A_i is the set of actions for boat i where $A_i \in \{Join, Balk\}$
- The reward function is designed to decrease the idle time (i.e. the time spent waiting for the operator) and respectively the total mission time. Hence, it considers the effect of all actions as a measure of time (instead of keeping two objectives time and failure rate):
  - by joining the queue (joining will increase the team total waiting time)
  - by balking and fail (operator should spend more time to resolve the failure)

The above definitions for state and action are the same as the definitions in Section 6.2.2 for one boat. However, the model in this section uses a different reward function (i.e. the reward mentioned above instead of using a scalar value as in Section 6.2.2), and considers the effect of failure as a measure of time to simplify the learning objective of the team. In other words, here we focus on the total team reward and the idle time of the system (i.e. the time spent waiting for the operator), while in Section 6.2.2, we view the number of failures as an individual parameter (i.e. not a measure of time) that should be taken into account, when selecting an action.

In this model, by considering multi-robot scenario, the state of the system includes the state of all robots. In the above model, we did not explicitly consider the state of the queue (i.e. number of requests), as it can be generated by counting the number of boats which are in their $\mathbf{W}aiting$ state (i.e. $S_b = \mathbf{W}aiting$).

In general, there are two major approaches for learning in multi-robot scenarios [49]. The first approach is called team learning and uses a single learner to learn the behavior for the entire team. In contrast, the second approach uses multiple concurrent learners, usually one for each robot, where each learner tries to learn and improve its behavior. Each of these methods has its own advantages and disadvantages which make it preferable in different domains [49, 78]. In particular, the major problems with team learning approach are the explosion of the state

space (i.e. it keeps the states of the entire team), and the centralization of the learning approach that needs to access the states of all team members.

Using the team learner in our application, the state space will be very large which decelerates the convergence to the optimal value. For example, for 5 boats with the above state representation, the state space will include more than one million states, hence requiring a prohibitive long time to estimate the optimal strategies for each state and action permutations.

The main advantage of independent learners in our domain is that, this domain can be decomposed into subproblems (e.g. each boat holds its own state space) and each subproblem can be solved by one boat. In general, two main challenges arise in concurrent learning including, credit assignment and non-stationary dynamics of the environment (see Chapter 3 for more details.)

However, the problem (i.e. human-multi-robot interaction) that we are modeling as MARL has some special properties, that can be exploited to achieve an effective and efficient approach.

First, the action selection at each step (i.e. when an event happens) only requires one agent to select either to join or balk. This decision will only affect the decision of the future arrivals. This problem can be considered as a large single-agent reinforcement learning, where at each step only part of the system will be changed, and only one action must be selected according to that part. Hence, the reward can go directly to that agent. It is different from the situations, where all agents should decide at each step (i.e. joint actions), which results in the well-known credit assignment issue [2, 49].

However, when each boat considers only its local state without knowing the state of the queue, finding the optimal behavior for the team may become impossible and the model may compute lower quality solutions. Therefore, we add the state of the queue to the local state of each boat, and then we use independent learners approach. The team objective is to find proper strategies for better utilization of the queue.
To sum up, the three discussed models are the following:

Full State a team learner has access to the joint full state of all robots which is $S = S_1 \times S_2 \times ... \times S_N$. When an event happens to a boat, the action $\langle Join, Balk \rangle$ for the corresponding boat will be selected and the state of the system will be updated[3]. The Q-value of the team learner will be updated accordingly.

Local State - Unobservable Queue (LocalState-UQ) the second approach (i.e. independent learner) is used for each boat. Each boat observes only its local state $S_i = \langle S_b, N_{tasks} \rangle$ and will select an action accordingly. In this model, each boat updates its local Q-values interacting with the system and receiving the reward.

Local State - Observable Queue (LocalState-OQ) in this model, each boat in addition to observing its local state, has access to the size of the queue. The queue size shows the number of waiting boats inside the queue. The state repre-

[3] The update will only change the part of the state related to the corresponding boat.

sentation of each boat in this model is: $S_i = \langle S_b, N_{tasks}, S_q \rangle$.

The three models are different in their state representation, while the reward structure is the same for all of them.

The reward functions are as follow:

- $R(S_t = S_i, A_t = Join) = R_S - (N_{queue}\bar{\mu} + t_{serv})$
- $R(S_t = S_i, A_t = Balk) = R_F(\frac{\bar{\mu}}{\bar{\lambda}}) + N_{queue}$; *if* $S_{t+1} = F$
- $R(S_t = S_i, A_t = Balk) = R_T$; *if* $S_{t+1} = A$

where $\bar{\mu}$ and $\bar{\lambda}$ are average service time and arrival rate respectively. N_{queue} is the number of boats waiting in the queue, and t_{serv} is the average time needed to resolve the request. $R_S = 1$, $R_F = -2$ and $R_T = 0.3$ are application specific parameters that must be tuned empirically.

Finally, we use Q-Learning as the basis learning approach, while the same reward structure, same distribution functions for generating events and same distribution function for the service time are used for all three models.
In an independent learning setting, each robot interacts with the environment (i.e. selects an action), receives the immediate reward and updates its state-action values (i.e. Q-values) according to (6.2):

$$Q_i(s_i, a_i) \leftarrow Q_i(s_i, a_i) + \alpha(R_i + \gamma \max_{a\prime \in A_i} Q_i(s\prime, a\prime) - Q_i(s_i, a_i)) \tag{6.2}$$

where R_i and $s\prime$ are respectively the reward and the state observed by robot i after performing action a_i in state s_i ; $a\prime$ is the action in state $s\prime$' that maximizes the future expected rewards; α is the learning rate and γ is the discount factor.

Remark1 In the simulation, $\bar{\lambda}$ and $\bar{\mu}$ in the reward function are the same as λ and μ for generating the requests and service. However, for a field deployment, these two parameters should be estimated since we do not know the real arrival and service rate. In other words, for estimating these two parameters in a field deployment, we have to consider the average number of events being generated during an interval as *lambda* (e.g. 20 events have been generated in 1 hour (or 60 minutes)), and the average time spent to fix each request as $\frac{1}{\mu}$ (e.g. 5 minutes to fix each request, hence 12 events can be fixed in 60 minutes).

Remark2 The state of the queue, S_q, can be modified by robots' action (joining the queue) and the operator's action (leaving the queue). However, under the reasonable assumption that an arrival and a departure cannot happen exactly at a same time, only one entity can change the value of S_q at a time. Moreover, the possibility of having more than one event at the exact same time is very low. In particular, we assumed the time to change the state of the queue (transition time), is much lower than the time for a new event arrival. Under this assumption, even if two events happen within a short time interval, the first one will affect the state of the queue before the second arrives, hence the other robots will base their decisions on the updated queue size.

Remark3 With each event, only one boat makes a decision (i.e the others continue what they were doing). Hence, as mentioned before, credit assignment is not an issue, and the reward(or penalty) goes directly to the corresponding boat.

6.3.2 Empirical Evaluation

We perform several experiments in the water monitoring simulation to evaluate the behavior of the system. First, we train the robots following the three learning models introduced in Section 6.3.1 to compare their learning process including the total reward and convergence rate. After the learning phase, in the first set of experiments, we run 30 trials and we report the average results over all such runs. The goal is to show and compare the idle time and the total reward of different models. These set of experiments use the same estimation of λ and μ as the learning phase.

In the second experiment, our goal is to compare queuing models which allow the robot to balk and the model without balking option. To this end, we consider FIFO and SJF queuing models without balking, which are usual models used in the literature [20] and FIFO queue with balking property. The goal is to compare the team performance with respect to the idle time of the system.

Finally, the noise sensitivity of our main model *LocalState-OQ* is tested under different noise conditions and sources, including λ, μ and number of events.

Empirical Setup

The learning phase of balking models starts by performing the following steps:

- Operator defines a list of locations that should be visited and assigns each location (or a number of locations) to each boat. We consider 30 locations and 5 boats.
- Different event/request types, as in table 6.1, will be generated within an exponential distribution with parameter $\lambda = 0.25$.
- The operator's speed, for replying/resolving a request is chosen from an exponential distribution with parameter $\mu = 0.27$.
- In each episode of the learning phase, 20 events will be generated with the λ rate and boats select either to join the queue or not.
- A mission ends after the system encounters 20 events.
- For action selection in our model, we use ϵ*-greedy* method [4] with parameter $\epsilon = 0.1$.
- Our algorithm uses the learning rate $\alpha = 0.1$ and discount factor $\gamma = 0.9$ throughout the experiments [5].

According to the learning models presented in Section 6.3.1, we run the simulation to train 5 boats with the total number of 30 tasks to perform. Each episode

[4] Since, in our domain there are always two actions to select from, using *softmax* does not make any difference (see the definition of *softmax* in Chapter 3).

[5] These parameters have been tuned empirically.

of the learning phase starts with all boats in their Autonomy state (i.e. they do not need the attention of the human operator), then with arrival rate λ an event may happen to one boat.

Notice that, for all experiments we simulate the behavior of the operator by estimating the average service time needed for each request type (i.e. parameter μ).

Empirical Results on Behavior during the Learning Phase

Figure 6.2 shows the team rewards of each model, *FullState*, *LocalState-UQ* and *LocalState-OQ*, at each episode of learning phase. As we expected, the convergence rate of *LocalState-OQ* is much faster than the *FullState* while they both reach to a similar team reward. This is due to the larger state space of *FullState* which needs more iterations to estimate the value for each state and action. Comparing the team reward achieved in *LocalState-UQ* to *FullState* and *LocalState-OQ*, illustrates that knowing only the local state in a cooperative scenario, robots can make locally optimal decisions. However, *LocalState-OQ* and *FullState* have access to the essential data for making the decisions.

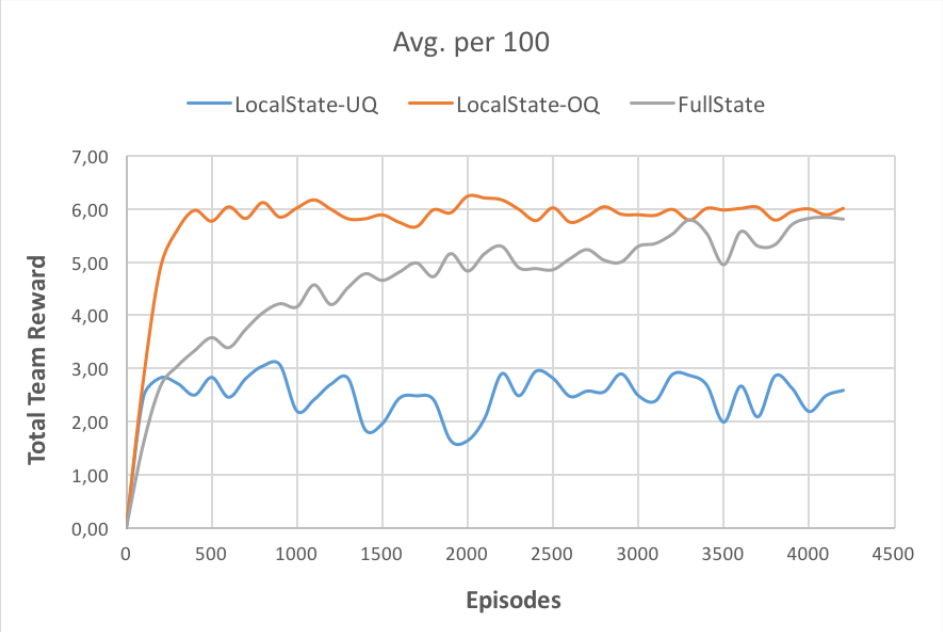


Fig. 6.2. Team accumulated reward in each episode of the learning phase (better viewed in color). The reward in each episode is the sum of the rewards of each independent learner (robotic boat).

Since, the reward given to each action is related to the parameters λ and μ, we expect our model to change its behavior by varying these two parameters.

Figures 6.3 and 6.4 show how *LocalState-OQ* will adapt to changes in parameter μ, where we increase and decrease its value by 40%. A sudden rise and drop happen respectively for each value, but then the system will converge to a stationary state.

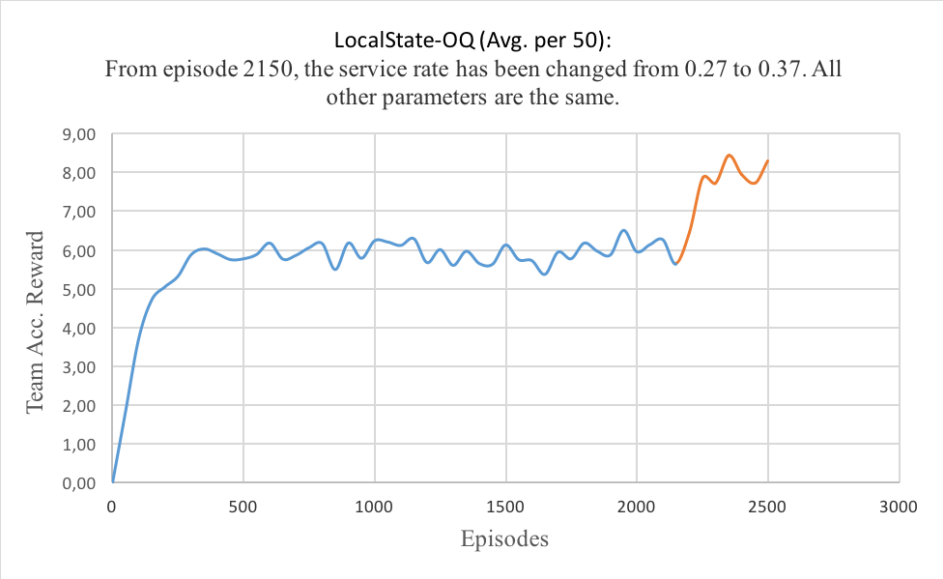


Fig. 6.3. Team accumulated reward in each episode of the learning phase (better viewed in color). From episode 2150, the service rate has been changed from 0.27 to 0.37. All other parameters are the same.

Figure 6.5 shows another experiment, where we vary the ratio of event types during the learning phase. The events were generated with a uniform distribution up to episode 2150. After that, as seen in Fig. 6.5, we consider the following situations:

- LocalState-OQ-H100: all events are from type 1 (E_1), which has the higher probability of failure (see table 6.1)
- LocalState-OQ-H80-ML20: 80% of the events are from type 1 (E_1) and the rest are uniformly distributed to E_2 and E_3
- LocalState-OQ-H50-ML50: 50% of the events are from type 1 (E_1) and the rest are uniformly distributed to E_2 and E_3
- LocalState-OQ-H10-ML90: 10% of the events are from type 1 (E_1) and the rest are uniformly distributed to E_2 and E_3
- LocalState-OQ-ML100: non of the events are from type 1 (E_1)

The results verify that, as there are more E_1 (i.e. high probability of failure), the system will gain less reward due to the increasing rate of failures. After several iterations, the learning curve becomes stationary and converges to a lower value than the time there are less E_1 requests.

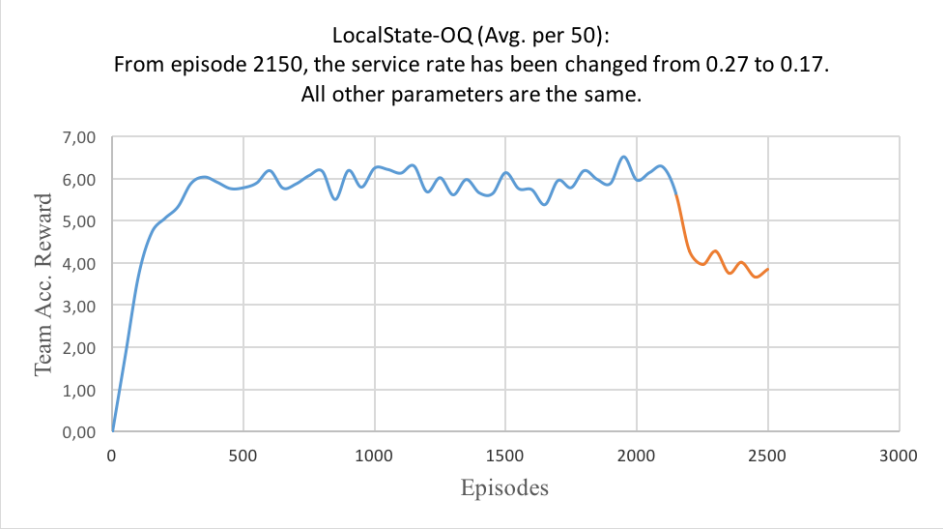


Fig. 6.4. Team accumulated reward in each episode of the learning phase (better viewed in color). From episode 2150, the service rate has been changed from 0.27 to 0.17. All other parameters are the same.

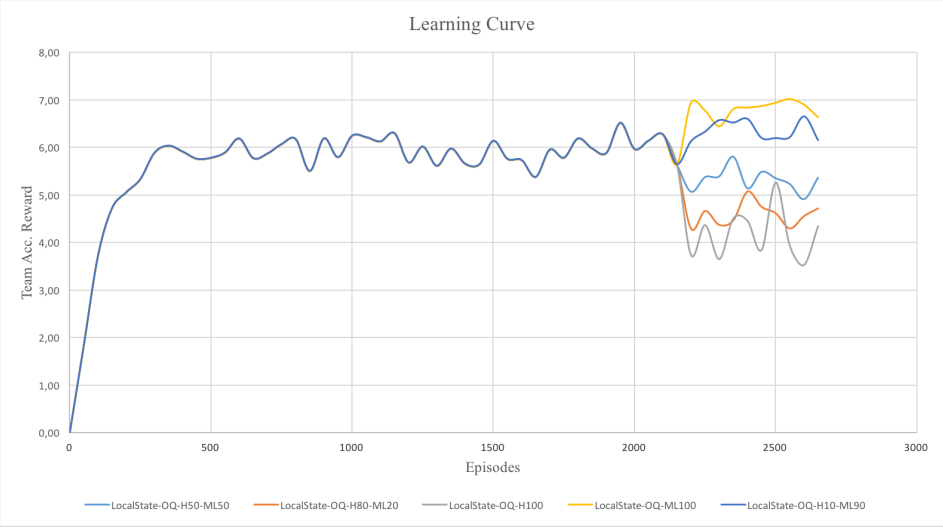


Fig. 6.5. Team accumulated reward in each episode of the learning phase (better viewed in color). After episode 2150, we vary the rate for each type of request. All other parameters are the same.

Empirical Results on Behavior during the Test Phase

After the learning phase, we run 30 trials on some test sets, using the learned values in Section 6.3.2. Figure 6.6 demonstrates the team reward for each learning models for 30 runs on the test set. A comparison on team reward between *LocalState-OQ* and *LocalState-UQ*, shows 56% gain for *LocalState-OQ*. Besides, a significant decrease (i.e. 40%) on average waiting time is shown in figure 6.7 when using *LocalState-OQ* rather than *LocalState-UQ*. One might expect the same reward value and idle time for *LocalState-OQ* and *FullState*. However, the results on Figures 6.6 and 6.7 show better performance values for *LocalState-OQ* than *FullState*. For example, the reward obtained during the test for *LocalState-OQ* is higher than the reward achieved for *FullState*. This difference is due to the fact that, *LocalState-OQ* model keeps only the size of the queue or S_q (i.e. it does not consider which boats are waiting in the queue), while *FullState* maintains the state of all boats which are in their **W***aiting* state (i.e. $S_b = \mathbf{W}aiting$). For example, whenever two boats waiting in the queue (assuming the other features of the state, such as severity are the same), *LocalState-OQ* will map the state to $S_q = 2$, while *FullState* will differentiate the states depending on which two boats are inside the queue. Since, the boats are homogeneous in our application scenario, *LocalState-OQ* results in better performance by abstracting away features that do not have a significant impact on the reward. This also makes *FullState* to converge slower than *LocalState-OQ*, due to the larger state space of *FullState* which needs more iterations to estimate the value for each state and action.

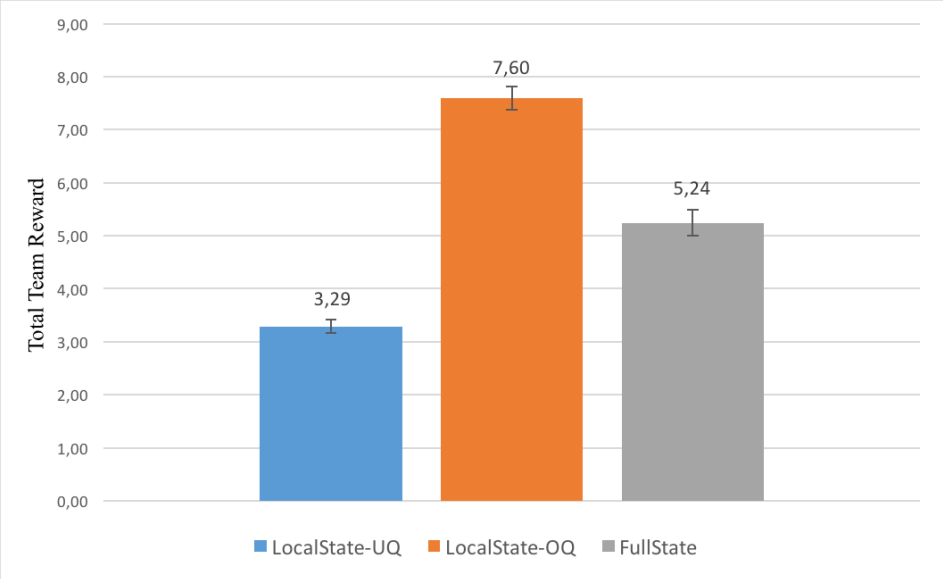


Fig. 6.6. Average accumulated team reward and the error bar according to the standard error of the mean are presented for three learning models: FullState, LocalState-UQ and LocalState-OQ.

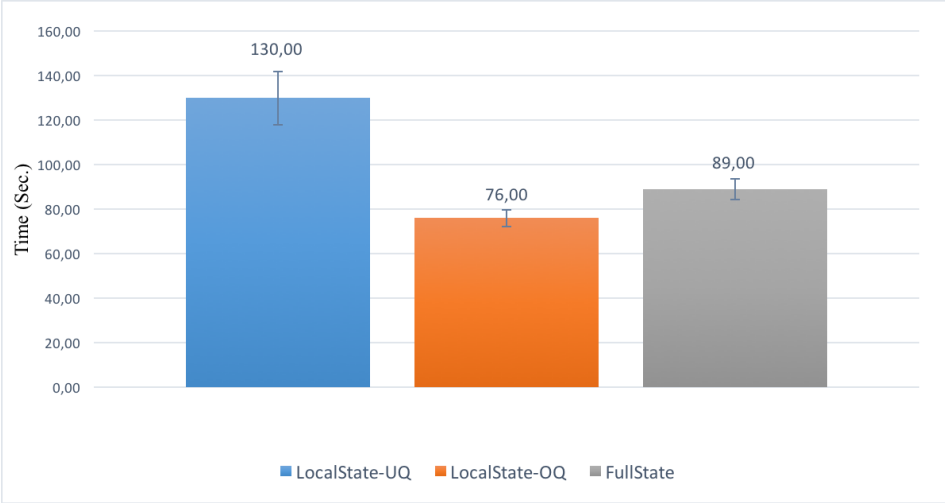


Fig. 6.7. Average team waiting time and the error bar according to the standard error of the mean for three learning models: FullState, LocalState-UQ and LocalState-OQ.

Next, we compare the behavior of queues with and without balking property (such as FIFO and SJF). For FIFO and SJF, we use the same event rate λ and service rate μ. In these two queuing models, boats always join the queue regardless of their request types and the queue size.

Figure 6.8 shows the team average idle time and the standard error of the mean for FIFO, SJF and three learning models. FIFO without balking, results in the worst queuing model, since boats wait for the operator until he/she becomes available.

In contrast, *LocalState-OQ* approach outperforms all other models. In more detail, it decreases the time up to 68% comparing to FIFO.

In general, the results in figure 6.8 indicates that, using balking models significantly decreases the idle time of the team even though, some events may result in failures. This is acceptable in our domain, since the penalties for failures are not critical but only result in a finite increase of time.

However, in some other domains, these failures may affect the performance of the team. So, in those domains the reward (or penalty) of failure should be defined in a way that considers the effect of failure.

Empirical Results on the Noise Impact during the Test Phase

To validate the noise sensitivity of our proposed model *LocalState-OQ*, we consider a set of experiments as follow.

First, we consider adding the same level of noise to both parameters λ and μ during the test phase. Figure 6.9 shows the team reward and Figure 6.10 shows the average idle time with the standard error of the mean for different levels of noise.

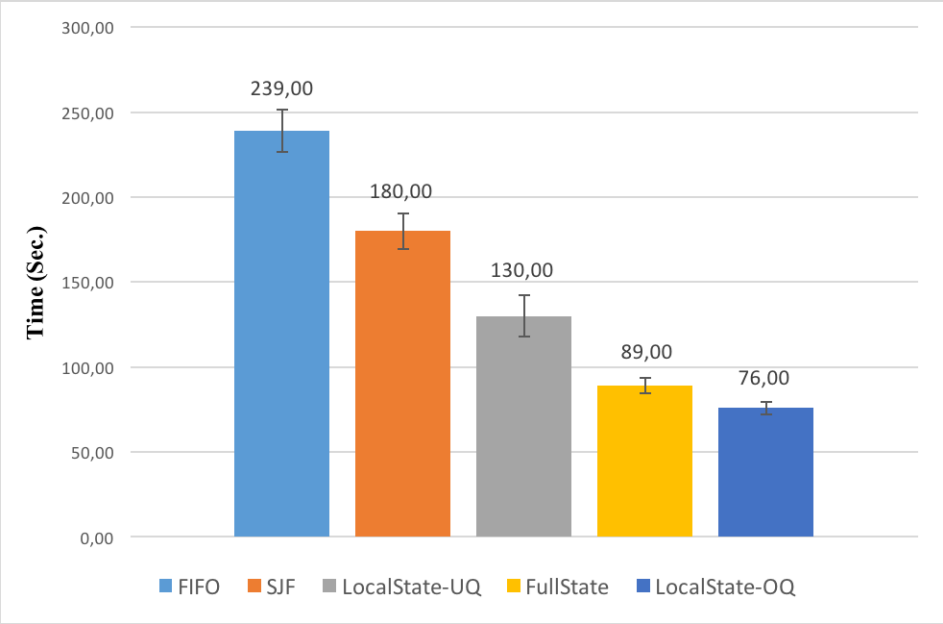


Fig. 6.8. Comparison between our balking model and queues without balking. The results show the average waiting time and the error bar of different models.

The results show that, the approach is able to cope with a significant amount of noise on both λ and μ. This behavior can be justified due to the fact that we consider a fixed number of events (i.e 20 events).

In another set of experiments, we consider inserting noise to each parameter separately (i.e. one at a time). For example, we use the $\lambda = 0.25$, same as the learning phase, and only vary the level of noise on μ (i.e. based on a uniform distribution). Figures 6.11 and 6.12 show the result.

We run a similar experiment, where $\mu = 0.27$ is fixed, while we change the level of noise on λ. Figures 6.13 and 6.14 show the results. The results show that, our model is more sensitive to the noise on mu. However, with the assumption of fixed number of events (e.g. 20 events), the model is less sensitive to the noise on parameter λ.

Next, we vary the number of events. Figures 6.15 and 6.16 show the team reward and idle time, when changing the number of events upto 35, and compare it with the 20 events used in the learning phase. When the number of events increases, the team reward will increase due to the extra actions (i.e. either join or balk). However, the idle time increases significantly, because of having more events.

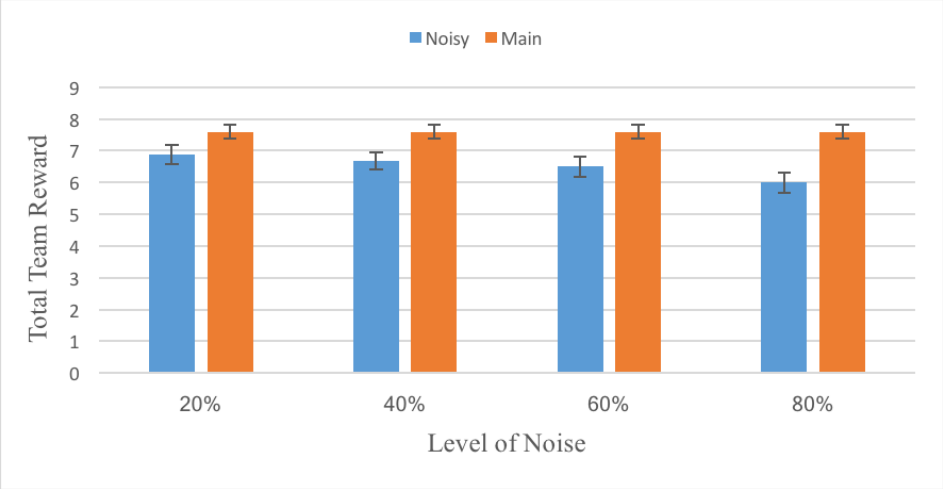


Fig. 6.9. Comparison of the team reward between LocalState-OQ (main) with different levels of noise on λ and μ (noisy). The results show the team reward and the error bar for each level of noise.

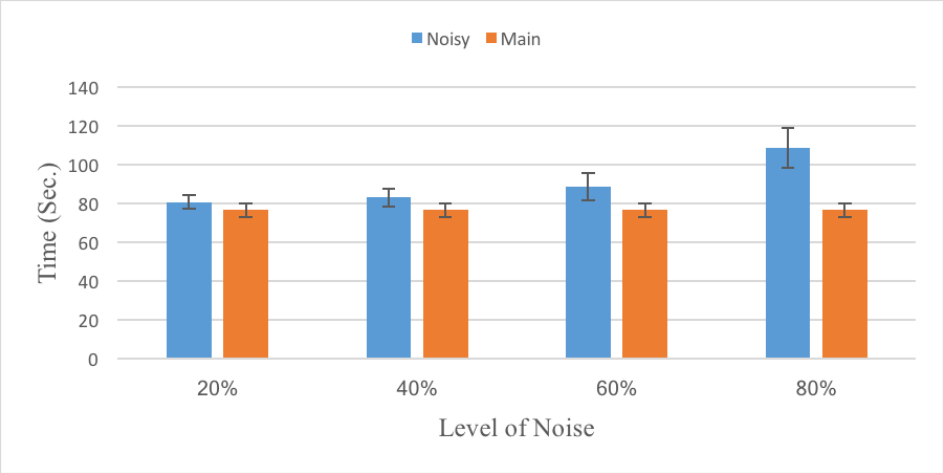


Fig. 6.10. Comparison of the idle time between LocalState-OQ (main) with different levels of noise on λ and μ (noisy). The results show the idle time and the error bar for each level of noise.

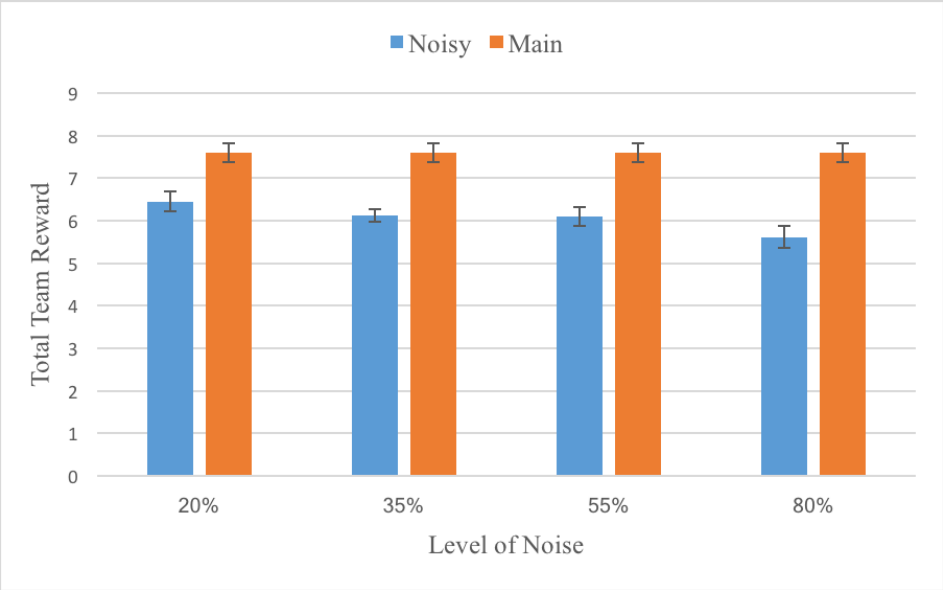


Fig. 6.11. Comparison of the team reward between LocalState-OQ (main) with different levels of noise on μ (noisy). The results show the team reward and the error bar for each level of noise.

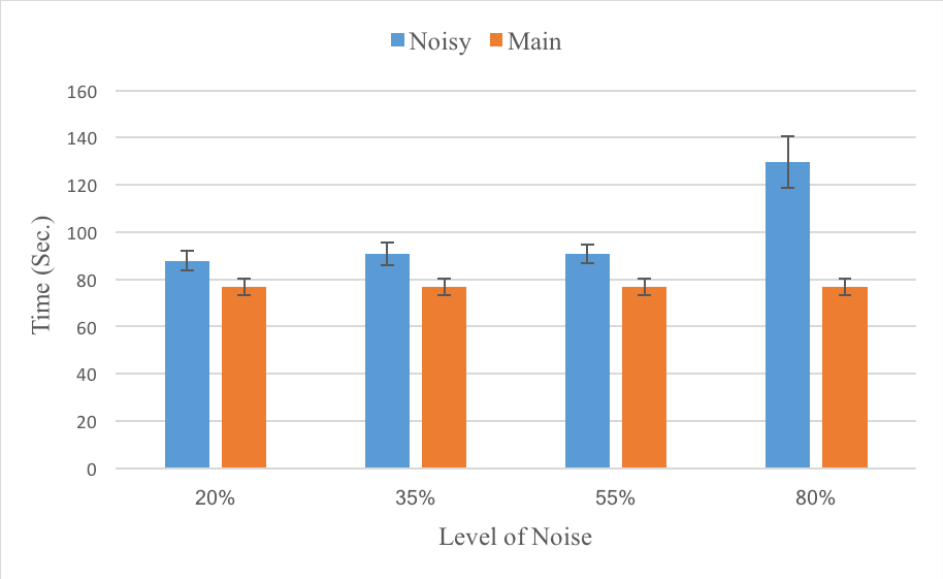


Fig. 6.12. Comparison of the idle time between LocalState-OQ (main) with different levels of noise on μ (noisy). The results show the idle time and the error bar for each level of noise.

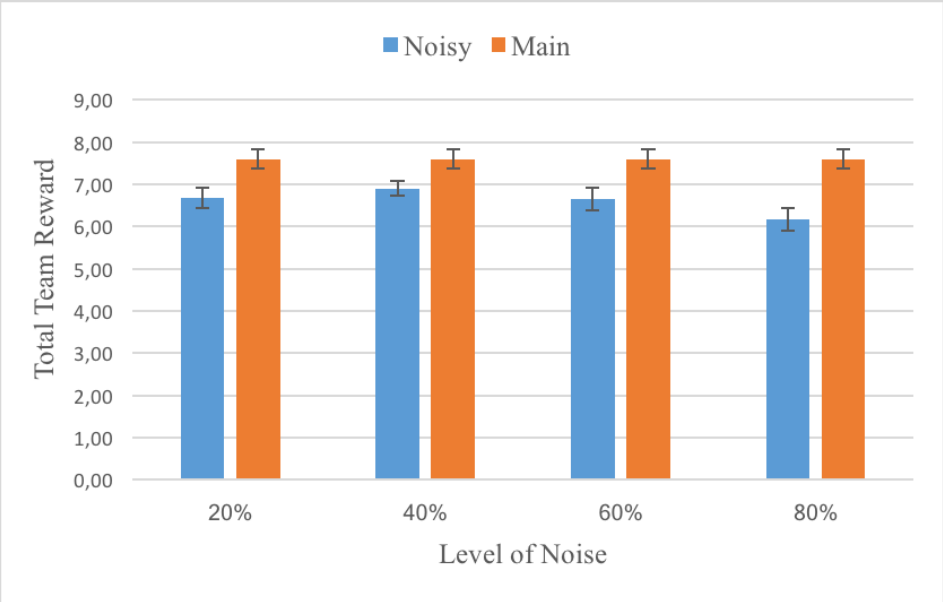


Fig. 6.13. Comparison of the team reward between LocalState-OQ (main) with different levels of noise on λ(noisy). The results show the team reward and the error bar for each level of noise.

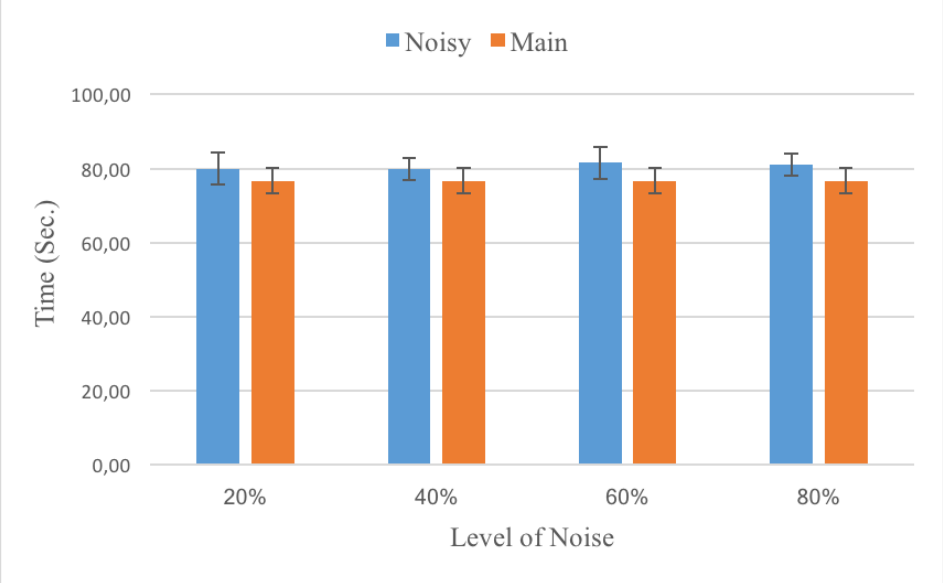


Fig. 6.14. Comparison of the idle time between LocalState-OQ (main) with different levels of noise on λ (noisy). The results show the idle time and the error bar for each level of noise.

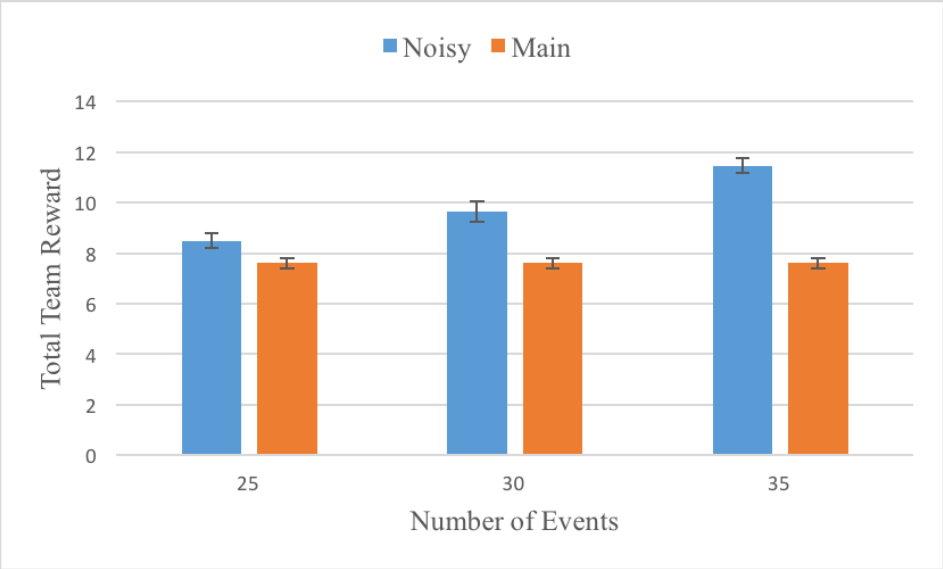


Fig. 6.15. Comparison of the team reward between LocalState-OQ (main) with different number of events (noisy). The results show the team reward and the error bar.

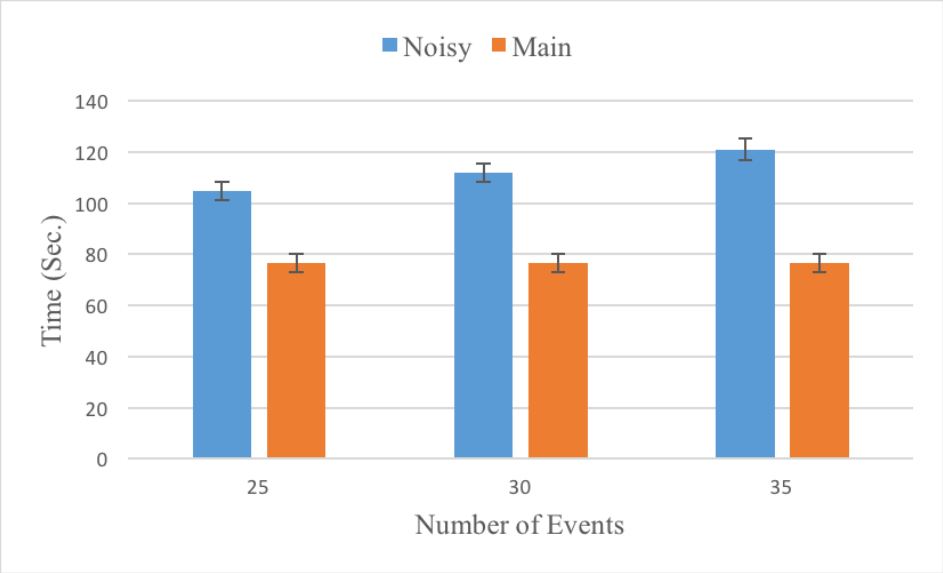


Fig. 6.16. Comparison of the idle time between LocalState-OQ (main) with different number of events (noisy). The results show the idle time and the error bar.

6.4 summary

We investigate the use of a queue with balking property to model human-multi-robot interaction in a water monitoring scenario. In general, we show that, using balking queues will significantly decrease the system idle time considering the balking cost.

Several approaches for computing the balking strategy are proposed and compared experimentally. First, we introduce a dynamic threshold policy by defining the reward and costs associated to the balking model for our specific water monitoring scenario. The empirical results show that, by using this dynamic threshold values, the waiting time decreases significantly compared to the queuing models such as FIFO and SJF. Furthermore, we apply Q-Learning approach to improve the balking strategies of the boats. The experimental results show that,our Q-learning approach compares favorably with FIFO and SJF queue models (in terms of waiting time) and it results in a lower percentage of failures with respect to the dynamic threshold model.

Then, we frame the problem as a Dec-MDP in which, each robot observes its local state and the state of the queue, and cooperates with other agents to optimize the use of the shared queue. We apply independent Q-Learning to find these cooperative strategies in our water monitoring multi-robot simulation. We compare the performance of our proposed model to two different models LocalState-UQ and FullState, and empirically show the significant improvement both on convergence speed and team reward. Furthermore, we perform experiments to show the effect of using balking queue on the idle time of the team. For this purpose, we compare queuing structures FIFO and SJF (without balking) with our balking model, and illustrate the notable decrease in idle time for the system.

7

Discussion

7.1 Summary

This thesis aims at enhancing the supervisory role of human in multi-robot applications. Within this context, we investigated two main research lines:

Smoothly Interrupting Team Plan We designed an interrupt mechanism that allows an operator to control the execution of a team plan by performing recovery behaviors or alternative plans. The proposed interrupt mechanism, which is based on CPN, allows a range of interrupts to be handled smoothly allowing the team to efficiently continue with its tasks after an operator intervene. Previous to this work, after such an interruption the operator would usually need to restart the team plan manually to ensure its success. We proposed two types of interrupts: a proxy interrupt that affects the execution flow of a subset of the platforms, and a general interrupt that specifies a particular recovery procedure for all the platforms.

We validated our approach considering an application of robotic watercraft. In more detail, we provided a quantitative evaluation of our interrupt mechanism by simulating the plan execution with and without the interrupts in a set of selected use cases. The empirical results show that, by combining the team-level and proxy-level interrupts, our mechanism provides a powerful and general model to allow sophisticated interactions between the human operators and team plans, resulting in a significant performance gain for the system. Moreover, we validated our approach on real platforms performing various experiments where a human operator should monitor and control the evaluation of several boats. Such experiments indicate that our mechanism can be of practical use in the actual deployment of robotic watercraft.

Balking Strategies for Human-Multi-Robot Interactions By assuming a team of robots capable of reflecting their needs, we modeled the human-multi-robot interactions as a *Balking Queue* where the robots decide whether to interrupt the operator or not. We developed three different decision making solutions that tell the robots whether to join or balk the queue. First we computed a threshold policy which is based on the dynamic features of the particular environment (i.e.

the reward of finishing a service and the cost of waiting). Then, we used single robot reinforcement learning approach to learn the balking policy of one robot while all other team members follow the dynamic threshold. In the last model, we consider a more general and realistic decision making framework, Dec-MDP, to model this problem. The Dec-MDP framework then has been solved by MARL approach (i.e. independent Q-learning method) to find the cooperative balking strategies. Notice that, in our single agent reinforcement learning, one robot adapts its policies towards the dynamic threshold of the rest of the team (i.e. biased to the dynamic threshold value), while in the proposed MARL, all robots learn simultaneously.

A simulation environment, based on a robotic water monitoring scenario, was then used to evaluate the performance of balking queue models. We compared the team performance of balking queue to queues without balking (e.g. FIFO and SJF). The results showed the notable decrease in the waiting time of the system (i.e the time spent by boat waiting for the operator).

7.2 Future Directions

Many possible future directions stem from this work. A first interesting direction is to extend the current plan specification framework to perform the analysis described in chapter 4 directly on the SPN (e.g., reachability analysis). Currently, we have to modify and convert the SPN plan in order to use it within CPN Tools or similar CPN Plan verification softwares. This modification is subject to the human errors, specially when the plan is large and complex. Therefore, developing a plan verification software compatible with SPN framework will ease the process of plan definition and validation.

Another interesting direction is to evaluate how difficult it is for a human designer to learn and use the SPN plan specification framework and the associated interrupt mechanism. Moreover, the current empirical work relies on modeling and evaluating the multi-robot system by simulating the operator's actions (e.g. number of click). We did not present a user study in this work to analyze the operator's performance using the proposed interrupt mechanism. A possibility, is to perform a user study to evaluate whether human operators (possibly with different backgrounds and education) can design SPN plans and use the interrupt mechanism efficiently.

Another promising direction for future work that touches upon similar issues is to consider the operator behavior in the learning process of the robots. This would allow to provide a more realistic model for service time, which is related to the operator's skills and speed. In more detail, a human study can illustrate how much the length of the queue affects the operator's efficiency and how much the diverse capabilities of the human operator can impact the learned policies of the robots.

Our short plan for expanding the current work includes the above directions.

One important future plan (probably in long term) would be to examine different MARL solutions. In the current work, mainly Chapter 6, the main goal is to present the concept of the balking queue for modeling human-multi-robot-interaction and not the learning approach. Hence, we apply the well-known Q-learning solution. We show the effectiveness of this approach in our simulation

water monitoring domain. However, the learned policies show some unpredictable patterns during the test under increasing uncertainties. To deal with this issue, we could investigate using other RL solutions specifically policy gradient methods or in general function approximation methods to better capture the uncertainty in both learning and testing phases. In particular, the policy gradient methods rely upon optimizing parametrized policies with respect to the long-term cumulative reward by gradient descent [51]. The main advantage of policy gradient in this kind of domains is that, the learned policy can be generalized from observed states to the unobserved states. This property of policy gradient methods (i.e generalization by predicting), not only makes them effective in high-dimensional spaces (e.g. we can learn in larger multi-robot systems), but also makes them to better perform in face of noise.

Moreover, instead of fixed policies for each state, the policy gradient can learn stochastic policies. Stochastic policies are in general more robust than deterministic policies, specially in partially observable states. When part of the state is hidden from the agent, a stochastic policy is more robust as it takes the uncertainty about inferring the hidden states into account.

For example, the main assumption of our approach is that the state of the queue is always accessible to the boats. However, robots may not be in constant communication with the base station (and the other robots). In this scenario, the action selection should be stochastic rather than a deterministic mapping from hidden state to actions. Hence, in this scenario, policy gradient methods could result in better performance than Q-Learning. Therefore, another area of future work is considering the model in which, the state of the queue is observable only during the learning phase. After learning, the state of the queue is not observable all the time. The solution for this problem is different from *LocalState-UQ* method, where the boats never access to the state of the queue.

Another possible direction is modeling the situations where the boats can decide to leave the queue when the expected waiting time, after joining the queue, does not meet their requirements. In this case, the action space of each robot includes $\{Join, Balk, Leave\}$.

Overall, this thesis provides a novel perspective to address the important and challenging problem of human supervisory control in multi-robot applications. The increasing use of multi-robot systems in real-world applications, gives rise to the need of supervision of human operator(s). Robots, specially at field sites, are often subject to unexpected events that can not be managed without the intervention of the operator(s). This thesis presents a mechanism to encoding how interrupts can be handled smoothly, allowing the team to efficiently continue with its tasks after an operator intervention. Moreover, the supervisory role of the human can present problems of overwhelming complexity. In this context, we show how the autonomy and self-reflection abilities of robotic platforms can be exploited to decide when and which requests must be sent to the operator, hence improving the performance of the system.

References

1. Christopher Amato, Girish Chowdhary, Alborz Geramifard, N. Kemal Ure, and Mykel J. Kochenderfer. Decentralized control of partially observable markov decision processes. In *CDC*, 2013.
2. Tucker Balch et al. Learning roles: Behavioral diversity in robot teams. *College of Computing Technical Report GIT-CC-97-12, Georgia Institute of Technology, Atlanta, Georgia*, 73, 1997.
3. Antonio Barrientos, Julian Colorado, Jaime del Cerro, Alexander Martinez, Claudio Rossi, David Sanz, and João Valente. Aerial remote sensing in agriculture: A practical approach to area coverage and path planning for fleets of mini aerial robots. *Journal of Field Robotics*, 28(5):667–689, 2011.
4. Raphen Becker, Shlomo Zilberstein, and Victor Lesser. Decentralized markov decision processes with event-driven interactions. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 302–309. IEEE Computer Society, 2004.
5. Raphen Becker, Shlomo Zilberstein, Victor Lesser, and Claudia V Goldman. Transition-independent decentralized markov decision processes. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 41–48. ACM, 2003.
6. Raphen Becker, Shlomo Zilberstein, Victor Lesser, and Claudia V. Goldman. Solving transition independent decentralized Markov decision processes. *Journal of Artificial Intelligence Research*, 22:423–455, 2004.
7. Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.
8. Richard Bellman. *Dynamic programming*. Courier Corporation, 2013.
9. Daniel S Bernstein, Christopher Amato, Eric A Hansen, and Shlomo Zilberstein. Policy iteration for decentralized control of markov decision processes. *arXiv preprint arXiv:1401.3460*, 2014.
10. Daniel S Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of markov decision processes. *Mathematics of operations research*, 27(4):819–840, 2002.
11. Bernard Berthomieu, Didier Lime, Olivier H. Roux, and François Vernadat. Reachability problems and abstract state spaces for time petri nets with stopwatches. *Discrete Event Dynamic Systems*, 17(2):133–158, June 2007.
12. Dimitri P Bertsekas. Dynamic programming and optimal control 3rd edition, volume ii. 2011.
13. Giuseppe Bevacqua, Jonathan Cacace, Alberto Finzi, and Vincenzo Lippiello. Mixed-initiative planning and execution for multiple drones in search and rescue missions.

In *Proceedings of the Twenty-Fifth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'15, pages 315–323. AAAI Press, 2015.

14. Daan Bloembergen, Karl Tuyls, Daniel Hennes, and Michael Kaisers. Evolutionary dynamics of multi-agent learning: A survey. *J. Artif. Intell. Res.(JAIR)*, 53:659–697, 2015.
15. Craig Boutilier. Sequential optimality and coordination in multiagent systems. In *Proceedings of the 16th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'99, pages 478–485, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
16. Lucian Buşoniu, Robert Babuška, and Bart De Schutter. *Multi-agent Reinforcement Learning: An Overview*, pages 183–221. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
17. Alan Carlin and Shlomo Zilberstein. Value-based observation compression for decpomdps. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '08, pages 501–508, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.
18. J. Casper and R.R. Murphy. Human-robot interactions during the robot-assisted urban search and rescue response at the world trade center. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 33(3):367–385, June 2003.
19. Sonia Chernova and Manuela Veloso. Interactive policy learning through confidence-based autonomy. *J. Artif. Int. Res.*, 34(1):1–25, January 2009.
20. Shih Yi Chien, Michael Lewis, Siddharth Mehrotra, Nathan Brooks, and Katia P. Sycara. Scheduling operator attention for multi-robot control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, Vilamoura, Algarve, Portugal, October 7-12, 2012*, pages 473–479. IEEE, 2012.
21. Philip R Cohen and Hector J Levesque. Teamwork. *Special Issue in cognitive Science and Artificial Intelligence*, pages 487–512, 1991.
22. John Collins, Corey Bilot, Maria Gini, and Bamshad Mobasher. Mixed-initiative decision support in agent-based automated contracting. In *Proceedings of the Fourth International Conference on Autonomous Agents*, AGENTS '00, pages 247–254, New York, NY, USA, 2000. ACM.
23. Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg. *Lectures on concurrency and Petri nets: advances in Petri nets*, volume 3098. Springer, 2004.
24. Gregory A. Dorais, R. Peter Bonasso, David Kortenkamp, Barney Pell, and Debra Schreckenghost. Adjustable autonomy for human-centered autonomous systems. In *on Mars, in First International Conference of the Mars Society*, 1998.
25. Alessandro Farinelli, Nicoló Marchi, Masoume M. Raeissi, Nathan Brooks, and Paul Scerri. A mechanism for smoothly handling human interrupts in team oriented plans. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '15, pages 377–385, Richland, SC, 2015. International Foundation for Autonomous Agents and Multiagent Systems.
26. Alessandro Farinelli, Masoume M. Raeissi, Nicolo' Marchi, Nathan Brooks, and Paul Scerri. Interacting with team oriented plans in multi-robot systems. *Autonomous Agents and Multi-Agent Systems*, 31(2):332–361, March 2017.
27. Francesco Maria Delle Fave, Alex Rogers, Zhe Xu, Salah Sukkarieh, and Nick Jennings. Deploying the max-sum algorithm for coordination and task allocation of unmanned aerial vehicles for live aerial imagery collection. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 469–476, May 2012.
28. Terrence Fong, Charles Thorpe, and Charles Baur. *Robot as Partner: Vehicle Teleoperation with Collaborative Control*, pages 195–202. Springer Netherlands, Dordrecht, 2002.

29. Terrence Fong, Charles E. Thorpe, and Charles Baur. Robot, asker of questions. *Robotics and Autonomous Systems*, 42:235–243, 2003.
30. Claudia V. Goldman and Shlomo Zilberstein. Optimizing information exchange in cooperative multi-agent systems. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '03, pages 137–144, New York, NY, USA, 2003. ACM.
31. Boris Gromov, Luca M. Gambardella, and Gianni Di Caro. Wearable multi-modal interface for human multi-robot interaction. In *in: IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR2016.*, pages 240–245, 10 2016.
32. Carlos Guestrin, Shobha Venkataraman, and Daphne Koller. Context-specific multi-agent coordination and planning with factored mdps. In *AAAI/IAAI*, pages 253–259, 2002.
33. James P. Gunderson and Worthy N. Martin. Effects of uncertainty on variable autonomy in maintenance robots. In *IN WORKSHOP ON AUTONOMY CONTROL SOFTWARE*, pages 26–34, 1999.
34. Eric Horvitz, Andy Jacobs, and David Hovel. Attention-sensitive alerting. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, UAI'99, pages 305–313, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
35. Ronald A Howard. Dynamic programming and markov processes. 1960.
36. M. Ani Hsieh, Anthony Cowley, James F. Keller, Luiz Chaimowicz, Ben Grocholsky, Vijay Kumar, Camillo J. Taylor, Yoichiro Endo, Ronald C. Arkin, Boyoon Jung, Denis F. Wolf, Gaurav S. Sukhatme, and Douglas C. MacKenzie. Adaptive teams of autonomous aerial and ground robots for situational awareness. *Journal of Field Robotics*, 24(11-12):991–1014, 2007.
37. Kurt Jensen. *Coloured petri nets: A high level language for system design and analysis*, pages 342–416. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.
38. Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Publishing Company, Incorporated, 1st edition, 2009.
39. Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3):213–254, Jun 2007.
40. Gal A. Kaminka and Inna Frenkel. Flexible teamwork in behavior-based robots. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*, AAAI'05, pages 108–113. AAAI Press, 2005.
41. Gal A. Kaminka and Inna Frenkel. Integration of coordination mechanisms in the bite multi- robot architecture. In *Proceedings of 2007 IEEE International Conference on Robotics and Automation*, pages 2859–2866, 2007.
42. Michael Lewis, Shi-Yi Chien, Siddarth Mehortra, Nilanjan Chakraborty, and Katia Sycara. *Task Switching and Single vs. Multiple Alarms for Supervisory Control of Multiple Robots*, pages 499–510. Springer International Publishing, Cham, 2014.
43. Borhen Marzougui, Khaled Hassine, and Kamel Barkaoui. A new formalism for modeling a multi agent systems: Agent petri nets. *JSEA*, 3(12):1118–1124, 2010.
44. Francisco S Melo and Manuela Veloso. Decentralized mdps with sparse interactions. *Artificial Intelligence*, 175(11):1757–1789, 2011.
45. T MURATA. Petri nets: properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
46. P. Naor. The regulation of queue size by levying tolls. *Econometrica*, 37(1):15–24, 1969.
47. Illah R. Nourbakhsh, Katia Sycara, Mary Koes, Mark Yong, Michael Lewis, and Steve Burion. Human-robot teaming for search and rescue. *IEEE Pervasive Computing*, 4(1):72–78, 2005.

48. Frans Adriaan Oliehoek and Matthijs TJ Spaan. Tree-based solution methods for multiagent pomdps with delayed communication. In *AAAI*, 2012.
49. Liviu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. *Autonomous agents and multi-agent systems*, 11(3):387–434, 2005.
50. Donato Di Paola, Annalisa Milella, Grazia Cicirelli, and Arcangelo Distante. An autonomous mobile robotic system for surveillance of indoor environments. *International Journal of Advanced Robotic Systems*, 7(1):8, 2010.
51. J. Peters. Policy gradient methods. *Scholarpedia*, 5(11):3698, 2010. revision #137199.
52. James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
53. Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.
54. David V Pynadath and Milind Tambe. The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research*, 16:389–423, 2002.
55. David V. Pynadath, Milind Tambe, Nicolas Chauvat, and Lawrence Cavedon. *Toward Team-Oriented Programming*, pages 233–247. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.
56. Masoume M. Raeissi, Nathan Brooks, and Alessandro Farinelli. *A Balking Queue Approach for Modeling Human-Multi-Robot Interaction for Water Monitoring*, pages 212–223. Springer International Publishing, Cham, 2017.
57. Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. Cpn tools for editing, simulating, and analysing coloured petri nets. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*, ICATPN'03, pages 450–462, Berlin, Heidelberg, 2003. Springer-Verlag.
58. Ariel Rosenfeld, Noa Agmon, Oleg Maksimov, and Sarit Kraus. Intelligent agent supporting humanmulti-robot team collaboration. *Artificial Intelligence*, 252(Supplement C):211 – 231, 2017.
59. Stephanie Rosenthal and Manuela Veloso. Mobile robot planning to seek help with spatially-situated tasks. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, AAAI'12, pages 2067–2073. AAAI Press, 2012.
60. Paul Scerri, Balajee Kannan, Pras Velagapudi, Kate Macarthur, Peter Stone, Matt Taylor, John Dolan, Alessandro Farinelli, Archie Chapman, Bernadine Dias, and George Kantor. *Flood Disaster Mitigation: A Real-World Challenge Problem for Multi-agent Unmanned Surface Vehicles*, pages 252–269. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
61. Paul Scerri, David V. Pynadath, Nathan Schurr, Alessandro Farinelli, Sudeep Gandhe, and Milind Tambe. *Team Oriented Programming and Proxy Agents: The Next Generation*, pages 131–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
62. Paul Scerri, David V. Pynadath, and Milind Tambe. Towards adjustable autonomy for the real world. *J. Artif. Int. Res.*, 17(1):171–228, September 2002.
63. Matthias Scheutz and James Kramer. Reflection and reasoning mechanisms for failure detection and recovery in a distributed robotic architecture for complex robots. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 3699–3704. IEEE, 2007.
64. Jürgen Schmidhuber. Realistic multi-agent reinforcement learning. In *Learning in Distributed Artificial Intelligence Systems. Working Notes of the 1996 ECAI Workshop*. Citeseer, 1996.
65. Sven Seuken and Shlomo Zilberstein. Memory-bounded dynamic programming for dec-pomdps. In *Proceedings of the 20th International Joint Conference on Artifical*

Intelligence, IJCAI'07, pages 2009–2015, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

66. Dominik Sieber, Selma Music, and Sandra Hirche. Multi-robot manipulation controlled by a human with haptic feedback. In *IEEE/RSJ Conference on Intelligent Robots and Systems (IROS)*, pages 2440–2446, 2015.
67. Adrian Stoica, T Theodoridis, Huosheng Hu, Klaus McDonald-Maier, and David Barrero. Towards human-friendly efficient control of multi-robot teams. In *Proceedings of the 2013 International Conference on Collaboration Technologies and Systems, CTS 2013*, pages 226–231, 05 2013.
68. Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.
69. K Suzanne Barber, Anuj Goel, and Cheryl E Martin. Dynamic adaptive autonomy in multi-agent systems. *Journal of Experimental & Theoretical Artificial Intelligence*, 12(2):129–147, 2000.
70. Milind Tambe. Towards flexible teamwork. *J. Artif. Int. Res.*, 7(1):83–124, September 1997.
71. Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.
72. Craig Tovey, Michail G. Lagoudakis, Sonal Jain, and Sven Koenig. *The Generation of Bidding Rules for Auction-Based Robot Coordination*, pages 3–14. Springer Netherlands, Dordrecht, 2005.
73. Prajna Devi Upadhyay, Sudipta Acharya, and Animesh Dutta. Task petri nets for agent based computing. *INFOCOMP Journal of Computer Science*, 12(1):24–35, 2013.
74. Abhinav Valada, Prasanna Velagapudi, Balajee Kannan, Christopher Tomaszewski, George Kantor, and Paul Scerri. Development of a low cost multi-robot autonomous marine surface platform. In *Field and Service Robotics*, pages 643–658. Springer, 2014.
75. Jijun Wang and M. Lewis. Human control for cooperating robot teams. In *Human-Robot Interaction (HRI), 2007 2nd ACM/IEEE International Conference on*, pages 9–16, March 2007.
76. Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.
77. Dianxiang Xu, Richard Volz, Thomas Ioerger, and John Yen. Modeling and verifying multi-agent behaviors using predicate/transition nets. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 193–200. ACM, 2002.
78. Ping Xuan and Victor Lesser. Multi-agent policies: From centralized ones to decentralized ones. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 3*, AAMAS '02, pages 1098–1105, New York, NY, USA, 2002. ACM.
79. Jieyu Zhao and Jurgen Schmidhuber. Incremental self-improvement for life-time multi-agent reinforcement learning. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cambridge, MA*, pages 516–525, 1996.
80. V. A. Ziparo, L. Iocchi, Pedro U. Lima, D. Nardi, and P. F. Palamara. Petri net plans. *Autonomous Agents and Multi-Agent Systems*, 23(3):344–383, Nov 2011.