Dynamic Resource and Task Management

Markus Klinik Jan Martin Jansen Fok Bolderheij

March 2018

Abstract

Carrying out maritime missions comprises many phases from preparation to execution. In the long term, we would like to have an integrated toolchain that supports the crew at every phase. In this paper, we study concepts for resource and task management in the execution phase. When the tasks to be executed have been identified, the question arises who should be assigned to them. This is both a scheduling and an assignment problem. We narrow down what kind of problem we have at hand to get an understanding what a first step towards an integrated command and control system could look like. This also enables us to classify our problem with the existing literature on planning and scheduling. We develop a domain model for tasks and resources, their connection via capabilities, together with assessment functions to compare assignments. We study what kind of information would be needed to give useful scheduling advice.

Keywords

command and control, mission management, online task scheduling, online resource assignment, workflow modelling

1 Introduction

In previous work (Bolderheij, 2018; Kool, 2017) we developed prototype Command and Control applications for reasoning about mission goals and tasks to achieve the goals. These applications dealt with assignments of resources to tasks. We studied examples from the maritime domain, that is navy ships with their tasks and resources. The concepts of tasks and resources were defined with the maritime domain in mind, which led to definitions that were too domain specific. In this paper we want to generalize some of these concepts, and then specialize them back to the maritime domain. Our concepts should be general enough to be specialized to a variety of different scenarios, and to different levels of granularity inside one scenario.

The core goal of this paper is the concept of a system that provides interactive decision support for on-line scheduling for command and control on board of navy ships. With this in mind, we define the concepts of tasks and resources, together with a way for tasks to specify their resource requirements. We describe how resources specify which tasks they can execute. We define a quality metric to determine which resources are most suitable for executing a task. Resources can be subject to degradation, which allows modelling broken machines and tired people.

These definitions serve as basis for a literature study to identify techniques that are useful to us. We demonstrate how the system would handle an example scenario.

The long term vision of this work is a tool for command and control that takes workflows from a planning phase as input, and finds suitable assignments of resources to tasks so that they can be performed efficiently. A supervisor monitors execution of tasks and stays in contact with the crew to solve problems when they occur. The supervisor has overview over the current state of affairs, running tasks, busy resources, and remaining capacities. Crew members carry some kind of smart device with an app that displays all tasks they are currently assigned to, in the form of a checklist. They can communicate with the supervisor by marking tasks as *in progress*, *paused*, *done*, or maybe *impossible to progress*. Tasks that require more complex interactions can have an extended user interface that allows, for example, entering data. The app can notify crew members when they are assigned to high priority tasks.

2 A Model For Tasks and Resources

The main goal of this work is the development of a model for tasks and resources that is abstract enough to be applicable to a variety of situations and granularities.

In this paper we talk about three kinds of roles, the modeller, the planner, and the supervisor. The modeller identifies tasks, resources, and capabilities in the domain of discourse and formalizes them in our system. Such a formalization is called a *model*. The planner uses the model to solve the problems of a particular scenario. Planning involves picking the right tasks to achieve a goal and assigning the right resources to do so efficiently. Once planning is done and execution of a plan starts, the supervisor monitors progress of the plan and makes adjustments to it as necessary.

These three roles are not necessarily performed by different people, nor do they happen at distinct phases. Modelling and planning can go hand in hand, as can planning and supervision.

2.1 Running Example: Making Pizza

We illustrate the design decisions of this paper using a recurring example. The example is simple, but shows many of the issues we want to be able to handle. The example serves to check that our system is expressive enough to model a common scenario, to discuss granularities where it makes sense to stop modelling, and to get an idea about what conclusions can be drawn from some given information. The example will be developed further in subsequent sections, and discussed in more detail in section 5.

Imagine a family planning their dinner. The family consists of four people, Mom, Dad, Alice and Bob. They decide that they want to make pizza. For the sake of our discussion, making pizza consists of two tasks, preparing the dough and finishing the pizza. Our family has enough ingredients at home to prepare the dough, but for the toppings someone needs to shop groceries. The family owns a car and a bike, but only Mom and Dad have a driver's license.

2.2 Tasks and Resources

At the heart of our system lies the assignment of resources to tasks. A *task* is a unit of work that we want to be able to plan and manage. A *resource* is a scarce, uniquely

identifiable or quantifiable object that a task must exclusively claim in order to be executed.

The important aspect here is that resources must be able to be claimed exclusively. One goal of our system is reasoning about resource conflicts, and conflicts can not occur when something can be infinitely shared. For example people, hammers, matchsticks, fuel, and storage space can be resources in our model, but information or time can not.

Georgievski and Aiello (2015) argue that time is a consumable resource. We argue that time cannot be regarded as a resource at all, because time lacks the exclusivity that is essential for something to be considered a resource. Consider time and fuel. If we have five liters of fuel and two tasks that require four liters each, then these two tasks can neither be executed in sequence nor in parallel. If we have a deadline five minutes from now then we can execute as many four-minute tasks as we want in parallel, provided that all other resource requirements are met. Furthermore, time cannot be saved up. If we do not use some fuel now, we still have it later. Time is gone once it has passed, no matter whether we execute a task in it.

We do not differentiate between *performers*, which are the resources that actually perform tasks, and passive resources like tools and materials.

2.3 Capabilities and Assignments

When a modeller wants to specify that a task needs some resource, he usually does not specify a particular individual, but rather describes it in abstract terms. For example, the modeller might specify that making pizza requires a cook, but he would not immediately specify that it specifically requires Bob. Furthermore, Bob might have many more skills besides making pizza. To express this indirect coupling between tasks and resources, we introduce the concept of *capabilities*. Capabilities are descriptions of the roles of the resources that a task requires. In turn, resources specify all the roles they are capable of fulfilling. In case modellers want to specify a particular resource, they can do so by using a capability that only one resource has.

Capabilities also always specify a capacity. This way modellers can specify that a task needs a certain quantity of some resource, for example five liters of fuel. There is a difference between one two-capacity capability and two one-capacity capabilities. For example, if a task needs two mechanics, the modeller has to specify two one-capacity capabilities. If the modeller specifies one mechanic with capacity two, then this requirement can never be met, because people have a capacity of one for their capabilities.

During planning, once the desired tasks have been chosen, a scheduler can consider capabilities and some optimization criterion to find an optimal assignment of particular resources to the tasks. Resources that are assigned to a task are said to be *claimed* by the task.

Figure 1 shows a UML class diagram for tasks, resources, and capabilities, and the relations between them. A task can require many capabilities, a resource can have many capabilities, and a task can claim many resources. A resource can be claimed by many tasks if its capacity permits it.

Another aspect we want to be able to capture is partial dependencies between tasks. Sometimes tasks depend on other tasks, which constrains the order in which they can be executed. Dependencies express that a prerequisite task has to be completed before another one can be started. The possibility to model such dependencies is represented

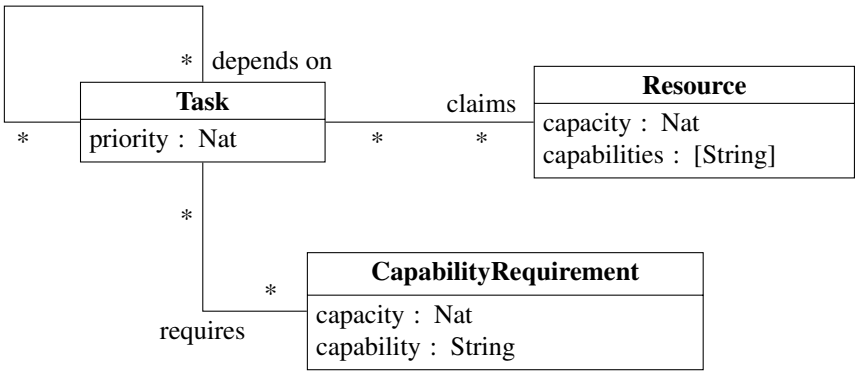

Figure 1: UML class diagram for Tasks, Resources, and Capabilities

in Figure 1 by the relation from Task to Task. One task can depend on many other tasks, and one task can be prerequisite for many other tasks.

3 Capability Functions

If there are multiple candidate resources that can be assigned to a task, a decision must be made which resource to pick. In order to do this, we would like to have a quality measure that estimates how good a resource will perform a task.

We express quality as a percentage.

Quality does not only encode the skill level of a resource, but can be used to encode all kinds of preferences. Our system uses *capability functions* to determine the quality with which a resource can execute a given task. Whether degradation should be factored directly into quality or requires independent handling requires further investigation.

The result of a capability function is a percentage. To understand what the input should be, consider the following examples.

Example 3.1. *On board of a marine ship are two radars. The task is to track a helicopter that has already been detected. Tracking means having high-frequency, high-accuracy updates of the helicopter's position.*

Example 3.2. *A gas pipe is on fire in room A1 on board of a ship. The task is to extinguish the fire. There are two methods available to extinguish the fire: a water mist installation and a dry chemical extinguisher.*

Example 3.3. *For having dinner, someone must shop groceries. On the shopping list is, among other things, some fresh fish. The fishmonger on the weekend market is five kilometers away. There are a bike and a car available.*

In order to determine, for each scenario, which resources are best suited to execute the task, we must take into account not only the tasks and the resources, but also the helicopter, the fire, and the shopping list.

The quality of a radar track depends on the type of the target to be tracked, it's distance to the ship, and the weather. The quality of extinguishing a fire depends on the type of the fire and who is closest to the location of the fire. The quality of a person and a vehicle to do shopping depends on the items on the shopping list.

In each case, the signature of the capability function must be different, in both the number of arguments and their type. This makes it hard to construct a generic scheduler that, without domain knowledge, can assess the quality of assignments in different scenarios.

In order to have uniform capability functions, we introduce two new concepts, targets and systems. A *target* is the reason why a task must be executed. The targets in our examples are the contact of type helicopter, with it's position and speed, the fire with its location and type, and the shopping list with its contents. Other examples for targets are customer complaints, software bugs, or expected threats.

A *system* is a combination of resources which jointly provide a capability. Systems are needed when the quality of a resource can not be determined in isolation. Systems have two attributes: the capability they provide and a list of capabilities they require to do so. An assignment of concrete resources to the required capabilities of a system is called a *system assignment*. Single resources can act as systems, in cases where the quality of a single resource can be judged in isolation.

In example 3.3, it is the combination of driver and vehicle whose suitability for shopping must be judged, not driver and vehicle in isolation. Mom with the car is well suited, while both Bob or Mom with the bike are less suited, and Bob with the car is not suited at all, because Bob doesn't have a driver's license.

In example 3.2 we have two ways of achieving the same goal. The fire can be extinguished by the water mist installation, or by a person with a fire extinguisher. This can be modelled as a task that requires a single capability *fire brigade*, and two systems that provide this capability. The first system requires a capability *water mist*, while the second system requires two capabilities *fire fighter* and *fire extinguisher*.

Many capability functions also require access to the *environment*, a database with the current operational picture. The environment contains things like the weather conditions and the topology of the surrounding area.

All these considerations suggest the following signature for capability functions.

Definition 3.4. *(Capability function) The quality of a system for a task depends on the system assignment, the task, the capability the system provides for the task, the task's target, and the environment. The expected quality is given as a percentage.*

$$capability : (SystemAssignment, Task, Capability, Target, Environment) \rightarrow Percentage$$

3.1 Extensibility of the Scheduler

As modellers model scenarios by creating tasks, resources, targets, and systems, they also have to create capability functions. To provide the algorithmic expressiveness some capability functions need, they are best specified in a programming language. This requires modellers to be at least somewhat skilled in programming.

There are three common ways to let users specify input with algorithmic content. First we could provide a small domain specific language, specifically tailored for specifying capability functions. Second, we could make the system extensible by providing an interface to a scripting language like Python or Lua. Third, we could let modellers use the same language the system itself is implemented in.

The first option requires substantial implementation effort to provide features of a general purpose programming language, and even more if modellers want access to external libraries. The second option still requires some implementation effort to make the interface of the scheduler available to the scripting language. For a first prototype

implementation, the third option is the best way, as it makes a general purpose programming language available to modellers at very low implementation overhead, as all interfaces and data structures can readily be accessed.

4 Human in the Loop

When a resource becomes degraded during execution of a plan and some tasks can no longer be finished, a change in plan is required. Dealing with unexpected situations often requires flexibility, creativity and improvisation, all abilities where humans perform better than computers. On the other hand, in order to make informed decisions, humans need insight into the current state of affairs. Information needs to be distributed quickly and stored accurately, and big numbers of possibilities must be assessed in a short time, all jobs where computers outperform humans.

We fall into line with Johnson (2014) and Bradshaw et al. (2013) who argue that for effective human-robot collaboration, full autonomy of the team members is not desirable: "Resilience in human-machine systems benefits from a teamwork infrastructure designed to exploit interdependence". He defines interdependence as the complementary relations that participants in a joint activity rely on in order to remedy their lacking capabilities.

At the core of Johnson's model for human-machine collaboration lies the concept of OPD, which stands for observability, predictability, and directability. Observability means making relevant aspects of one's own status, knowledge and environment available to others. Predictability means that others can rely on their prediction about one's own behavior when planning their actions. Directability means the ability to influence the behavior of others, and being influenced by others.

For our application, observability means that the ship needs an up-to-date picture of the degradation of systems and people on board. The other way around, the ship's internal picture should be accessible to the supervisor. Predictability means that in similar situations, the system should come up with similar plans, and small changes in the environment should lead to small changes in a plan. Directability means that the supervisor can influence planning and always override any decision the system makes. For example the supervisor should be able to make plans that are nonsensical to the system, or start tasks whose resource requirements are not fully met.

We propose two categories of actions which allow supervisors to intervene in the execution of a plan. The first category is called *plan B* and includes decisions that can be made before a plan is executed. The second category is called *dynamic planning* and includes actions to modify a plan while it is being executed.

4.1 Plan B

A plan B is an alternative plan to achieve the same goal, using different tasks and resources. In the extreme case, plan B uses completely different tasks and resources. Realistically, there is some overlap between plans A and B.

If plans A and B overlap at some resource, and this resource becomes degraded, then B also can no longer be executed. Consequently, for an alternative plan to be useful, an estimation of *expected degradation* is required. Plans should not overlap in resources that are likely to degrade. Our system should give insight into resource overlap.

The advantage of alternative plans is that they can be activated quickly. Coming up with them however requires additional planning beforehand.

4.2 Dynamic Planning

Dynamic planning allows supervisors to change plans while they are being executed. The system should give supervisors the freedom they need to improvise.

There are two reasons why a dynamic change in plan becomes necessary. First, a resource required for execution of a running task becomes degraded. Second, a new goal is identified, and new tasks must be executed to achieve it.

When a running task becomes disabled because one of its assigned resources degrades, the supervisor has the following options.

- Keep the task running, but change its resource assignment. This is useful for tasks that allow the new resources to pick up the work where the degraded resources left it.
- Cancel the current task, start new tasks and assign different resources to them. Cancelling is useful for tasks that can not be handed over to other resources.
- Pause parts of plan A, start new tasks with new resources to repair the degradation, then continue with plan A.

Both pausing and cancelling a task frees all its assigned resources.

When additional tasks must be started because new goals have been identified, supervisors need similar actions. Goals have priorities, and tasks have priorities derived from them. A *resource conflict* exists if a set of new tasks cannot be executed because there are not enough resources available. Supervisors must find ways to resolve such conflicts, and our system must give them the tools to do so. This includes querying the current status to find solutions, and modifying the current status to implement them. For example, given a new task, the system displays all tasks with lower priority such that when they are paused, their freed resources allow the new task to be executed. The supervisor can then pick some running tasks and either cancel or pause them to free their resources.

5 Making Pizza

In this section we look at a situation in the daily life of a four-person family. They are sitting at the breakfast table on a Saturday morning, planning their dinner. We walk through the planning process while identifying relevant tasks and resources. The goal of this study is to make sure that our system is expressive enough to describe the scenario. Furthermore, we discuss the granularities where it makes sense to stop modelling.

The family consists of four people, Mom, Dad, Alice and Bob. The family decides that they want to have pizza for dinner today. This suggests that there is a task *make pizza*.

5.1 Making Pizza

Whenever we identify a new task, we must consider two questions. First, does it make sense to split the task into subtasks? Second, which resources are required to execute

the task?

Making pizza consists of many subtasks, as is evident by reading any pizza recipe, and we could go even further and specify subtasks down to the level of individual hand movements. This is of course too fine grained, and even the individual steps in the recipe are too detailed for the purpose of planning dinner. The goal of planning is to provide actors with the necessary instructions to perform their tasks independently.

Being too specific is not useful, and so is being too general. Instead of having a task *make pizza*, it is conceivable to have a generic task *cook meal* that is parameterized by a recipe. The recipe would then determine which subtasks there are and which ingredients and appliances are required to execute the task. At this point, we might as well take the recipe itself as the task, which would make the *cook meal* task useless.

For our scenario we assume that the family members are sufficiently skilled in cooking so that the instruction *make pizza* is enough for them to know what to do. There is however one subtask we do want to make explicit. Preparing the dough is a subtask that can reasonably be performed independently and even by a different person.

For the sake of discussion, let's say that there are enough ingredients for the dough, but not for the topping. The family decides that one person can start making the dough, while another person shops for the remaining ingredients. After both these tasks are completed, the pizza can be finished. This suggests three tasks in total: *make dough*, *shop groceries*, and *finish pizza*. The task *finish pizza* depends on both *make dough* and *shop groceries*.

These three tasks have the right granularity that allows performers to execute them without further instructions.

5.2 Required Resources

The next step in the modelling phase is to identify for each task what resources it needs. The goal is to be reasonably realistic and specify only those resources that are relevant for identifying conflicts.

All three tasks need a person to execute them, and we have to specify the required capabilities. There are many possibilities of how precise we want this specification to be. For example, we could be very precise and model that for making dough, we need someone with the capability *make dough*. We could also model that we need a *cook*, or even that we just need a *person*. In this example, we go with the capability *cook* for both tasks *make dough* and *finish pizza*. For the task *shop groceries* we say that we need a *shopper*.

When considering the required materials and tools for the task *make dough*, we again have to think about the right precision. Do we explicitly model every required kitchen appliance, including the oven and individual spoons, or do we just specify that a functional kitchen is needed? Do we specify all the ingredients in the recipe? The answer is again guided by the conflicts we want the system to be able to detect. On the one hand, more information means the system can draw more conclusions. On the other hand, the more precisely we specify the resource requirements of a task, the more work it is to model the resources themselves, and to be useful this information must be kept in sync with the real world.

For our scenario, we do not specify the ingredients, but we specify that a functional kitchen is required. We do the same for the task *finish pizza*, and specify that it needs a cook and a kitchen.

At this point, the modeller has to be careful. We just specified that two tasks need the same resource, a kitchen, and if we assume that only one kitchen exists, our system

Name	Required capabilities
make dough	cook, kitchen
shop groceries	shopper, vehicle
finish pizza	cook, kitchen

Table 1: Tasks

Name	Capabilities
Mom	cook, shopper
Dad	cook
Alice	cook, shopper
Bob	shopper
Bike	vehicle
Car	vehicle
Kitchen	kitchen

Table 2: Resources

does not allow these tasks to be executed in parallel. In this particular case that is not a problem, because the second task depends on the first one anyway, but there might be situations where we want two tasks in parallel in the same kitchen. Sometimes two people can work in one kitchen on different tasks, so the kitchen is not exclusively claimed by either. In this sense, the kitchen does not qualify as a resource according to our definition. To capture the situation in our system, the modeller could split the kitchen into a number of shares, and specify that the cooking subtasks require one share each.

The task *shop groceries*, in addition to a shopper, also requires a vehicle. We assume that the family has two vehicles, a bike and a car. When assigning resources to this task, not all combinations of shoppers and vehicles make sense, because only Mom and Dad have a driver's license. Modellers can exclude impossible combinations of resources with capability functions, which are discussed in section 3.

With the information gathered so far, the model is precise enough for the purpose of this example. Tables 1 and 2 summarize the identified tasks and resources. We could have been more precise, and also modelled that *shop groceries* requires money, or that *finish pizza* requires ingredients for the topping. Again, this comes down to the detail of the instructions the system should give to the people involved.

5.3 Scheduling

With the information gathered about the scenario so far, it is possible to construct schedules. A *schedule* consists of two parts. First, a *workflow*, which is a complete specification of the order in which to execute tasks. Second, an *assignment* of resources to the tasks in the workflow.

The three tasks of our example and their dependencies permit three possible workflows. They are shown in Figure 2. Workflow A specifies that *shop groceries* and *make dough* must be executed in parallel. Parallel execution for us means that they are to be started at the same time and that they cannot share any resources. Workflows B and C specify that all three tasks have to be executed in sequence. In both cases, the same person can be assigned to all of them. With our resources and the requirements for the tasks, there are 42 possible assignments for workflow A, and 54 for each of B and C, which makes a total of 150 ways of making pizza.

Without additional information for assessing schedules, a scheduler can do little more than list all possibilities. A scheduler could optimize for least total resource usage, which means it tries to reuse resources as much as possible, for example by assigning the same person to all three tasks in B and C. We, however, would like to

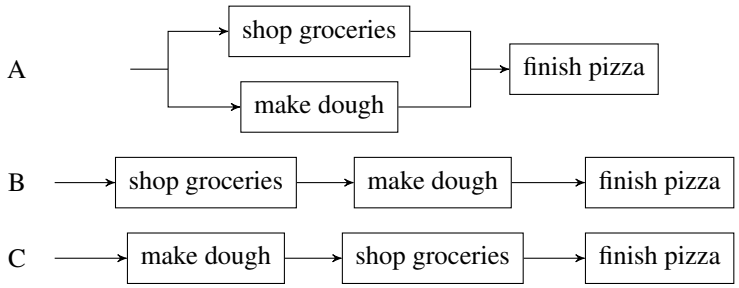

Figure 2: Three ways of making pizza

have an objective function that tells which schedules are the best, according to some optimization criteria.

When estimations of the durations of tasks are available, these criteria can be ones that have been studied extensively in scheduling literature, like maximum completion time or sum of completion times.

For now, we want to focus on the quality with which a resource can execute a task, rather than durations. In this example we choose workflow A, with Mom and the car as best suited for shopping, while Dad prepares the dough and finishes the pizza, because he has the most experience with these tasks.

5.4 Execution

With responsibilities settled, the family begins execution of the tasks. Mom drives with the car to the supermarket, while Dad starts making the dough.

On the way to the supermarket, the car breaks down. In our system, this is modelled by marking both Mom and the car as degraded. Now that the goal of having pizza for dinner can no longer be reached, alternative solutions must be found.

The task *shop groceries* must be stopped, as it does not make sense to see it as a task that can be taken over by other resources. This frees the resources Mom and car, but as they are degraded, they will not be considered when looking for alternatives. When starting a new instance of *shop groceries*, the system offers two possible assignments: Alice with the bike, or Bob with the bike. However, we still have Mom waiting in the broken car, and getting her home has higher priority than making pizza. Dad knows how to get the car running again, because he has repaired the same problem before. The solution to the problem looks like this: Dad grabs his tools and takes the bus to repair the car. Alice takes his place in the kitchen and continues preparing the dough where Dad left off. Bob takes his bike and shops groceries.

In order to implement this solution in our system, the system must grant a supervisor the following possibilities.

1. The resources Mom and car must be marked as degraded.
2. The task *shop groceries* must be cancelled.
3. A new task *repair car* must be created. This task has not been conceived in the planning phase, so the supervisor must have the ability to create new tasks on the fly.
4. The task *make dough* must be paused, so that Dad becomes free for the repair task.
5. Alice is assigned to *make dough*, and the task can be resumed.

6. A new instance of *shop groceries* is started, with Bob and the bike as resources.

The supervisor must perform steps 1 to 3 by hand, while the system offers advice for steps 4 to 6. In particular, the system should figure out that when the high-priority task *repair car* pops up, there still is a solution to make pizza.

6 Related Work

Kuhn (1955) showed that the assignment problem can be solved in polynomial time with the Hungarian method. The assignment problem asks to distribute n people to n jobs, where each person is differently qualified for each job, such that the total quality is maximized. Our assignment problem is a bit different however, because in general we require multiple resources per task, and when tasks run in sequence the same resources can be assigned to different tasks.

The literature on machine scheduling has results about machines that can process different kinds of jobs, which are called *multi-purpose machines*. It appears that our concept of resources can be seen as multi-purpose machines. Scheduling with multi-purpose machines and arbitrary processing times is NP-hard (Brucker et al., 1997).

Task that require multiple resources seem similar to what is known as *multiprocessor tasks* in the scheduling literature. Scheduling tasks that require two dedicated processors is already strongly NP-complete (Kubale, 1987).

Seeing that we have a combination of the assignment problem with multi-purpose machines and multiprocessor tasks, it appears that we have to resort to heuristic methods to find solutions. A promising method has been described by Mencía et al. (2015), using genetic algorithms to find schedules for tasks with precedence and skilled operators.

7 Future Work

The two most important next steps are further literature study and a minimal viable prototype implementation. We already know that finding exact solutions to our scheduling problem is infeasible, so we have to see what kind of techniques exist for heuristically finding okay solutions. After hopefully finding such a technique, and most probably adapting it, we have to make an implementation which allows us to run example scenarios. This lets us verify that we are on the right track, that the data model covers the relevant aspects of our domain, and that the answers we get are interesting and have the potential to become useful.

A user-friendly interface is not a goal for the prototype. In subsequent work it would be interesting to study how the differences between qualitatively similar solutions can be highlighted to the user. This is especially interesting when the choice of resources doesn't matter, for example when permutations of a set of resources all lead to solutions with identical quality. Furthermore we would like to find out how to present the consequences of choosing a particular solution for the remaining tasks.

8 Conclusion

In this paper we study the problem of resource allocation in command and control scenarios. We narrow down the kind of problem we want to solve to get an understanding

of how a first step towards an integrated command and control system could look like. We identify what kind of information such a system needs in order to provide useful answers.

The central components in our system are *tasks* and *resources*, and their connection via *capabilities*. Tasks require capabilities while resources provide capabilities. An abstraction we call *systems* can be used to express that a capability can only be provided jointly by several resources. Every system has a *capability function* that indicates the level of quality with which a given set of resources provides the system's capability.

Given a scenario with tasks, resources and systems, we want to solve a combination of an assignment and a scheduling problem. We want to find an ordering of the tasks together with an assignment of resources to the tasks such that the overall quality is as high as possible.

One goal of identifying the problem we want to solve is comparison with the scheduling literature. It turns out that the complexity of our problem makes exact optimization impossible. The best we can hope for is to find some form of heuristic that gives us a feasible solution if one exists and points out conflicts if no solution exists. This is acceptable, because in a realistic scenario, quickly finding some okay solution that gets the job done is more important than eventually finding the optimal solution.

Acknowledgments

We would like to thank the Manning and Automation Team at TNO, Rinus Plasmeijer, Bas van der Eng, and Terry Stroup for many hours of fruitful discussion.

This research is funded by the Royal Netherlands Navy and TNO.

References

Fok Bolderheij. A C2 framework for heterogeneous collaboration. *Manuscript submitted for publication*, 2018.

Jeffrey M. Bradshaw, Robert R. Hoffman, David D. Woods, and Matthew Johnson. The seven deadly myths of "autonomous systems". *IEEE Intelligent Systems*, 28(3): 54–61, 2013.

Peter Brucker, Bernd Jurisch, and Andreas Krämer. Complexity of scheduling problems with multi-purpose machines. *Annals of Operations Research*, 70(0):57–73, 1997. ISSN 1572-9338.

Ilche Georgievski and Marco Aiello. HTN planning: Overview, comparison, and beyond. *Artif. Intell*, 222:124–156, 2015.

Matthew Johnson. *Coactive Design: Designing Support for Interdependence in Human-Robot Teamwork*. PhD thesis, Technische Universiteit Delft, 2014.

Bram Kool. Integrated mission management voor C2-ondersteuning. Bachelor's thesis, Dutch Defence Academy, 2017.

Marek Kubale. The complexity of scheduling independent two-processor tasks on dedicated processors. *Information Processing Letters*, 24(3):141–147, 1987.

Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Res. Logist. Quart.*, 2:83–97, 1955.

Raúl Mencía, María R. Sierra, Carlos Mencía, and Ramiro Varela. Schedule generation schemes and genetic algorithm for the scheduling problem with skilled operators and arbitrary precedence relations. In *ICAPS*, pages 165–173. AAAI Press, 2015.