Task Monitoring and Rescheduling for Opportunity and Failure Management

José Carlos González
josgonza@inf.uc3m.es
Universidad Carlos III de Madrid (UC3M)
Leganés, Madrid, Spain

Manuela Veloso
mmv@cs.cmu.edu
Carnegie Mellon University (CMU)
Pittsburgh, Pennsylvania, USA

Fernando Fernández and **Ángel García-Olaya**
{ffernand, agolaya}@inf.uc3m.es
Universidad Carlos III de Madrid (UC3M)
Leganés, Madrid, Spain

Abstract

The CoBot robots, as other service robots, autonomously navigate in building environments performing different types of tasks that include item transportation and person guiding between locations. The CoBots can execute their planned routes, localize in the environment, avoid obstacles, and ask for help to humans to overcome their actuation limitations. However, they were not able to handle high-level unexpected events during execution, such as interruptions with new task requests that may need a careful analysis of rescheduling trade-offs. Unexpected events can be failures if their influence is on the pending tasks or opportunities if it is on the robot expectations. This work presents a new task-execution, monitoring, and rescheduling architecture, which includes a representation of new task features to be monitored to detect failures and opportunities, as well as a task scheduler to evaluate time and task features constraints. We demonstrate the new features in a task that needs to deliver hot coffee at some time, noting that the coffee gets cold with interruption delays.

Introduction

The development of autonomous service robots has been an important research topic in recent times. One notable example is the CoBot robots. CoBots are autonomous agents that exploit their presence to interact with people (Veloso et al. 2015). Their tasks involve physical movement from one place to another and different kinds of physical interactions with the real world, as to deliver a message or an object, to go to another place, to escort someone to an office, etc. The nature of these tasks is sequential and they cannot overlap.

Currently, a CoBot is basically a mobile robotic base with a computer, several sensors and a surface or basket to store objects. Any task that requires physical manipulation, like opening doors, pushing elevator buttons or getting objects, requires human help. CoBots are designed to ask for help to nearby people if they do not have the capability to perform a certain action of a task (Rosenthal and Veloso 2012). Tasks are created by the users using a web interface (Coltin, Veloso, and Ventura 2011) or just using the embedded touchscreen. Users can assign several time constraints to tasks. The robot has a task scheduler module in charge of finding a plan to perform all the operations. If the new task cannot be scheduled, the user is asked to change the parameters.

The scheduler can be considered as a high-level planner in a multi-level architecture like NAOTherapist (González, Pulido, and Fernández 2017). This hierarchical view of robotic architectures has been long discussed in the literature (Ghallab, Nau, and Traverso 2014). NAOTherapist uses a triple-layer planning mechanism where a high-level planner generates a plan to determine the tasks, or high level goals, to perform in a session (exercises to do). Traditionally, this high level works in an offline phase, without any rescheduling abilities. Then, a medium level controls and monitors the actions for each planned task to face unexpected events during their execution (movements of each exercise), sometimes even interrupting the current action. The low level acts like a path planner for each action, working directly with the robot platform.

CoBots can also replan their behavior in these medium and low levels of planning, avoiding obstacles, canceling a task if the target person is not in the place, etc. However, there are high-level events out of the control of these robots. One clear example arises when the robot has to deliver a spoken message to someone. It will try to find the recipient of the message in her office, but if, by chance, the person appears near the robot in its way, it will just continue driving to the office wasting the *opportunity* to deliver the message at that moment. The situation, in fact, will probably end up with a task fail since the person will not be at the office when the robot arrives. A *failure* also happens if the robot has to deliver an object, but loses it in the way. Therefore, CoBot was not able to tag these high-level events as opportunities or failures to avoid wasting time in unnecessary operations.

Being able to detect and act accordingly to some of these events is important to increase the autonomy of a robotic platform like CoBot. The automated planning field (Ghallab, Nau, and Traverso 2004) has approaches dealing with opportunities and failures. There are works about fast recovering of failures (Guzmán et al. 2015; Alcázar et al. 2010), which interleave planning and execution although they are more focused in the middle and low levels of planning. Other approaches use goal management techniques to take advantage

of some detected opportunities (Schermerhorn et al. 2009) by triggering soft goals at certain moments. Also, temporal planning has been used in opportunistic planning (Cashmore et al. 2017) approaches. However, the task scheduling problem addressed in this manuscript is heavily based on numbers and currently automated planning does not have good heuristics for numerical fluents.

Planning a schedule is a problem of mathematical optimization. There can be many feasible schedules for the same task pool with very different quality. The scheduler must try to find not only a suitable schedule, but also the optimal one in a limited amount of time. This time depends on the response time requirements of each application. All numeric variables used in CoBots can be restricted to integers without affecting the quality of the predictions, so Mixed Integer Programming (MIP) (Chen, Batson, and Dang 2010) is a useful technique for this work. This is the same technique used in the original CoBot scheduler (Coltin, Veloso, and Ventura 2011), but this work greatly improves the capabilities of that MIP model.

The main paper contribution is a novel architecture for a task scheduler that is able to detect and manage some opportunities and failures. It can also manage complex time constraints like delivering a coffee before it gets cold (cooldown time). Furthermore, this component can be reused in other multilayer architectures to control high-level events by defining only the interfaces and the available tasks types. With this scheduler, the high-level module gets also the ability to replan according to new high-level events, bringing it to an online phase allowing it changing its tasks and schedules dynamically.

High-level Task Scheduler Architecture

In CoBots, the scheduler is in charge of generating a valid schedule for the pool of remaining tasks and also managing their monitoring and execution to reschedule according to opportunities and failures. Figure 1 shows the different modules of the architecture of the developed component. Basically, users send tasks through an interface to the scheduler. Monitoring checks if a new task is valid by generating a MIP problem with some static data gathered from a shared knowledge base (for example distance among all involved locations). Then it sends this problem to an external solver which returns a valid schedule, if possible, in a certain amount of time. The tasks that are not executed yet are stored in the task pool. Afterward, the next task is sent to Execution when needed. Execution is just an interface for lower levels of planning of the robot. The robot constantly informs about the current state of the world. Monitoring gets these states and reasons if an opportunity or a failure has appeared, taking then the appropriate actions for each case. The rest of the CoBot works in a lower level, so the particular way it has to achieve each task is transparent for the scheduler.

High-level events can affect not only the movement or even the current task, but also future tasks in the schedule. These can be classified as opportunities and failures that must be detected by the robot in order to interrupt the execution if needed. They may not be easy to detect since they do not affect the execution of current medium or low-level

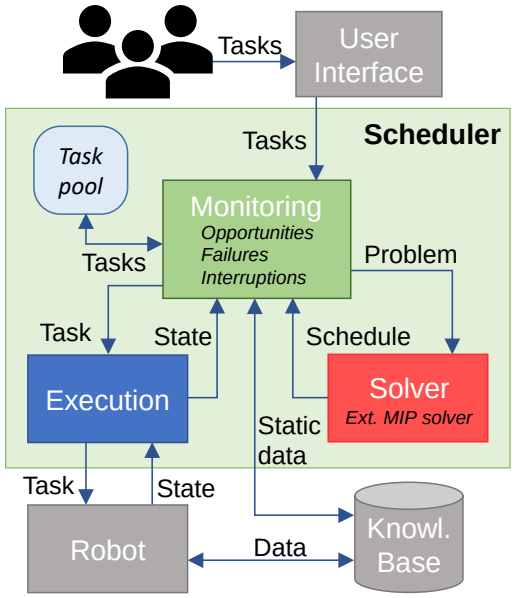

Figure 1: Architecture of the high-level task scheduler component.

actions. The need for the different modeled task parameters is motivated with a guiding example: delivering a hot coffee to someone.

Modeling of the Tasks

Tasks are modeled as a sequence of subtasks with different parameters. A subtask is a part of the task that can be interleaved with subtasks from other tasks. In CoBots, the size of the sequence is determined by the times that the robot must change the location sequentially to accomplish the task. For each task and subtask, several parameters must be specified, as described in Table 1, which shows the decomposition of the DeliverDrink task. The task decomposition and the parameters (task type, type owner, etc.) are specified by the designer of the robotic use case, so the architecture can reason with them. The meaning of each parameter is specified in the following paragraphs. The parameter set is enough for the example followed in this paper, but should be updated for different tasks and/or use-cases.

The developed graphic user interface (GUI) eases the insertion of high-level tasks like delivering a coffee. It procedurally decomposes each possible task type into the subtasks, according to the task description defined by the designer. Breaking them into parts permits to interleave subtasks of different tasks to improve the global performance of the executed schedule. Although delivering a coffee is used in this manuscript as an example, this mechanism can be used to model any other task that has a part that must be finished before a time threshold, and could be extended to more complex decompositions with three or more subtasks.

In the example, the user sets the time window in which the task can be executed, while the GUI gathers the information from a knowledge base to fill optional parameters,

		Task	Subtask-1	Subtask-2
User	Task type	DeliverDrink	MakeHotDrink	DeliverObject
	Task owner	Alice	*Alice*	*Alice*
	Location start	-	CoffeMaker	CoffeMaker
	Location end	-	CoffeMaker	AliceOffice
	Time start min	0	*0*	*0*
	Time end max	15	*15*	*15*
	Person target	Alice	-	Alice
	Object	HotCoffee	HotCoffee	HotCoffee
	Priority	-	10	10
Internal	Time operation	-	5	2
	Time cooldown	-	-	6
	Task depending	-	-	Subtask-1
	Opportunities	VIP	HotCoffee, *VIP*	Person target, *VIP*
	Failures	TO, BP	*TO, BP*	HotCoffee, *TO, BP*

Table 1: Task model instantiated in the decomposition of a DeliverDrink task type. Inherited parameters are in italic.

like location start with Alice's usual office and the nearest coffee machine. The GUI also fills internal parameters like an estimation of the operation time and the cooldown time. A subtask can only be executed after the subtask indicated in *Task depending* is finished. The task types used in this work are: go to location, deliver message, telepresence, attract attention, deliver object, deliver drink, escort someone, wait for emergency services and recharge battery.

Failures

A failure is a situation that impedes the success of the current task. It may require some actions to correct the problem or to abort the task if it is not feasible now. A *generic* failure can appear when a subtask cannot be completed because of blocked paths (BP) or a timeout (TO) due too many accumulated execution delays. Sometimes tasks have some *specific* failures. In the coffee example, the receipt could be indeed in his office, but the coffee could be stolen by someone from the basket. The internal parameter *Failures* indicates which other parameters must be invariant along the execution of the subtask. In Subtask-2 the robot must have *HotCoffee* in the basket at all times, as defined in Table 1.

Opportunities

In the DeliverDrink example, a *specific* opportunity appears when the robot is going to a room with the hot coffee in the basket and finds the target person in the corridor. It can try to deliver the coffee in that moment. This allows to finish the current task earlier and probably avoiding a future failure. The *Opportunity* internal parameter indicates which other parameters must be monitored along the execution of the subtask to detect potential specific opportunities, as shown in Table 1. A *generic* opportunity (valid for all tasks) can also improve the whole schedule. For instance, CoBot could identify a very important person (VIP) during the execution of the schedule. The robot could approach the VIP to greet him or just ask him whether it can help somehow. Spending time on this opportunity will delay the rest of the scheduled tasks, but the gain obtained by executing this important task could worth it if it has enough *Priority*.

Interrupting Tasks

Generic opportunities and failures must be defined in Monitoring for each application domain. For CoBots, a high priority VIP task is added in the task pool when the sensors identify a VIP in its surroundings. The internal predicates *Failures* and *Opportunities* control the specific high-level events to be detected for each subtask during all their execution.

Decomposing tasks in subtasks is very useful when interrupting high-level tasks. In the coffee example, if the robot already has the hot coffee in its basket and a VIP appears, Monitoring detects it as an opportunity. If the robot interrupts the coffee delivering to talk with the VIP, the coffee could be delivered cold. This is controlled with the *Cooldown time* internal parameter, which indicates the maximum time allowed after *Task depending* was finished.

In the DeliverDrink example, a change in the task pool (VIP) can force the robot to reason about the convenience of scheduling or not a certain task after making the coffee (CoffeeA) and before its delivery (CoffeeB). In this work there can be:

- **Redoing CoffeeA**. "I prefer to remake the coffee. The VIP can't wait, and I can deliver the coffee later"
- **VIP between CoffeeA and CoffeeB**. "I can spend some time talking with the VIP and then deliver the coffee on time"
- **VIP after CoffeeB**. "I can't redo the coffee later, but the VIP will be there for some time"
- **Cancel DeliverDrink**. "The VIP can't wait, and I can't redo the coffee later either. I prefer to cancel the coffee and talk with VIP"
- **Cancel VIP**. "The VIP can't wait, and I can't redo the coffee later either. I prefer to continue with my previous task and omit the VIP"

The result will depend on the gain that it will get after completing the schedule. This gain is modeled in the scheduler to allow the system to reason over these concepts.

Numerical Predictions for Tasks

The problems addressed in this work are deeply based on numbers. The constraints that involve the ordering of tasks depend mainly on the starting and ending time of them. The times indicated in a schedule are, in the best case, just a good prediction of their values. In particular, distance measures can also be expressed in time units because the average speed of the robot along each path is known, as well as the average time to perform certain operations like asking someone, preparing a drink, etc.

In fact, planning a schedule is a problem of mathematical optimization. There can be many feasible schedules for the same task pool with very different quality. The scheduler must try to find not only a suitable schedule, but also the optimal one in a limited amount of time. This time depends on the response time requirements of each application.

All numeric variables used in the scheduler can be restricted to integers without affecting the quality of the predictions, so Mixed Integer Programming (MIP) (Chen, Batson, and Dang 2010) is a useful technique for this work. It

allows finding solutions to numerical optimization problems thanks to mechanisms like simplex. This is the same technique that the original scheduler used (Blind Reference 3). A MIP solver receives the input data and uses a model which describes all needed mathematical constraints to find a valid schedule. The quality of the solution is evaluated using an objective function, so after finding a valid one the scheduler can continue the searching until it can demonstrate that the last was optimal.

In the MIP aspect, this work contributes with a new way to solve a kind of temporal problem (cooling down time) with it, taking priorities into account. The next sections explain the underlying architecture of the developed scheduler, detailing how this MIP model was created to obtain schedules that address the improvements mentioned here. They also describe the mechanisms and policies followed to detect opportunities and failures and how to manage the rescheduling processes they cause.

Modeling the MIP Problem

The scheduler has a pool of remaining tasks with different parameters. It requires a model to describe the constraints that must be fulfilled to obtain a valid solution. The model works with the input data provided, which in this case is the task pool. The solution returned is a valid schedule, if any. The variables that the solver tries to optimize are the starting s and ending e time for each task. It also optimizes the auxiliary binary variable $Previous(a, b)$ to calculate the optimal path according to the location of the previous task. The rest of the parameters is fixed by the input data. The formalization of the model is explained here with the symbols, the binary predicates and the constraints used. It is important to note that the distance measures described are modeled as predictions of how much time the robot needs to go from one location to another. All needed distances are precalculated with the help of the knowledge base before running the solver.

Positive integer parameters:
i, j, k: Any task of the pool
w^{min}: Minimum start time
w^{max}: Maximum end time
s: Start time (variable)
e: Ending time (variable)
o: Operation time
c: Cooling down time
p: Priority value higher than 0
l^s: Starting location
l^e: Ending location
$d(a, b)$: Distance (time estimation) between a and b

Binary parameters:
$Previous(i, j)$: Task i starts just before j (variable)
$Depends(j, i)$: Task j must start after i

Constraints:
$w_i^{min} \leq s_i \leq w_i^{max} - o_i - d(l_i^s, l_i^e)$
$w_i^{min} + o_i + d(l_i^s, l_i^e) \leq e_i \leq w_i^{max}$
$Previous(i, j) \Rightarrow s_i < e_i < s_j$
$\neg Previous(i, j) \Rightarrow s_i < s_k < s_j$
$Previous(i, j) \Rightarrow e_j \geq s_j + o_j + d(l_i^e, l_j^s) + d(l_j^s, l_j^e)$
$Depends(j, i) \Rightarrow e_i < s_j$
$Depends(j, i) \Rightarrow c_j \geq e_j - e_i$

Objective function:
$$\text{Minimize} \sum_{i=1}^{n} e_i p_i$$

The first two constraints restrict the value that the start and ending time variables can have to ease the searching process, taking the operation time and distance into account. The third constraint avoids the overlapping of tasks by stating that the starting time of a task must be lower than its ending time and that the ending time of a previous task must be lower than the starting time of any other one that follows it. The fourth one states that a task i cannot be previous of a task j if there is any other task k which starts between i and j. The fifth determines that the ending time must be greater or equal than the starting time plus the operation time plus the time needed to travel between the ending location of the previous task and the starting location of the current one and plus the time to travel between the starting and end locations of the current task. The sixth one controls the dependence of a task by stating that a task that depends on another must start after the other has finished. The last one states that the cooling down time must be greater or equal than the time between the ending of its dependence and the ending of the dependent task.

The model has an objective function to optimize the solutions. This function is the sum of the ending time of each task multiplied by their priorities. This guides the search to reduce the total time span of the schedule, but also to execute tasks with high priority earlier. A dummy initial task to take the initial location of the robot into account is used to improve this optimization.

When all constraints are met, a valid schedule has been found. However, this solution will probably not be optimal. The scheduler will continue searching for an optimal solution during a certain time threshold. If the optimal schedule is found, the solver returns it to start the execution of the first task when needed. If the solver cannot find an optimal solution in the given time, it returns a suboptimal solution.

The solver sometimes detects when a schedule is unfeasible, but if the problem is too hard, the solver could continue searching forever. If the solver cannot find a suboptimal solution in the given time, the schedule is considered unfeasible. This leads to different policies that are explained in the next section.

Before starting the search, the solver also checks if the input data are valid to avoid wasting time searching for an impossible solution. All integer variables must be positive, and priority higher than 0. Additionally, it considers these additional conditions.

Checks:
$w_i^{min} + o_i + d(l_i^s.l_i^e) < w_i^{max}$
$c_i \geq o_i + d(l_i^s.l_i^e)$

These checks can rely only on parameters of the own task. That is why they do not include the distance between the ending location of the previous task and the starting of the current one. That has to be calculated by the solver.

Monitoring

The Monitoring part receives partial states of the world from Execution, as shown in Figure 1. These states contain different elements like the current state of the task, the identity of the people near the robot or the objects placed in its basket. Monitoring receives this partial state and detects opportunities and failures for the task being executed, using the task definition (as the one shown in Table 1). Then, it decides whether it is necessary to run the MIP solver to take the possible generic and specific opportunities and failures into account and reschedule if needed. The outcome of this reschedule may cause the interruption of the current task. Monitoring sends the updated task pool to the MIP scheduler and receives a solution. Then it sends the first task to Execution.

There are two ways to add tasks to the pool. In the first one, the user sends a task to the robot through the user interface. The task is received, and the scheduler component tries to obtain a valid schedule. If it cannot, the new task is rejected, and the user must change the parameters of the task. In the second way, a task is generated internally by the robot in response to a detected opportunity. Tasks can be removed from the pool if a user aborts them or if they cannot be executed anymore because a failure. A task can finish earlier than expected due to a detected opportunity or a failure.

Opportunities can appear at any moment and may modify the current schedule. Generic internal parameter *Opportunities* of each task may indicate the existence of potential opportunities for the CoBot domain. Monitoring has a record of these parameters for all task and when something in the state of the world matches such parameters, a scheduled task can be modified or canceled, triggering then a rescheduling process. The outcome of this reschedule may cause the interruption of the present task. Currently all reschedules are done from scratch, without reusing previous solutions.

For example, DeliverDrink can have a person as a receiver. If the robot detects that person while it has the object in the basket, it can interrupt the current task to give the object to that person. This can happen with other tasks such as DeliverMessage and Escort. Similarly, if the robot has to give an object to someone and it detects that it already has one in its basket, the robot can deliver that object directly. In the same way, if the robot detects a very important person, Monitoring can add a new high priority task to the pool to reach him and perform a demonstration or give him information. The mechanism to determine if this new task can be scheduled or not is based in a gain metric explained in the next subsection.

Failures are similar to opportunities, except because they are unavoidable. Currently, CoBots control some lower-level failures like people not answering questions or not present to get some objects from the basket. Others like someone stealing an object from the basket that needs to be delivered, need the control of invariants along all the tasks of the pool. If an invariant parameter of a task changes in the state of the world (registered in the internal parameter *Failures*), the task is modified or even canceled, triggering a reschedule. Monitoring can also add tasks on failures to redo a chain of dependent tasks. Other possible failures are the need to charge the battery and reaching a certain delay threshold in which the planned starting time of the next task and the actual time is too much.

Rescheduling Policy

In response to a failure, Monitoring can remove tasks from the pool if the solver cannot find a suitable plan while rescheduling (it cannot redo a task because there is not enough time, for example). In this case, priority and amplitude of the time window are important parameters to cancel a task or not. The tasks that now do not have enough time to be completed are canceled directly. If, after this, the solver cannot find a plan, Monitoring starts canceling the next task with the lowest priority and the smallest time window that overlaps another. This process will continue until a schedule is found. When a task ends unexpectedly, the owner is alerted by email.

In the case of opportunities, if a task ends earlier than expected, Monitoring also triggers a reschedule to take advantage of it. If a new task is added into the pool (talk with the spotted VIP, for instance) and the scheduler cannot find a plan, then Monitoring evaluates the situation. The metric to determine this is the gain value g which is the sum of the priorities of the scheduled tasks.

Gain: $g = \sum_{i=1}^{n} p_i$

A VIP task has a very small-time window and very high priority. Canceling a DeliverDrink task to redo it later is a valid option if possible because more tasks can be done, and more gain will be obtained. The policy developed in this work tries to preserve the tasks already scheduled because it only considers redoing the current one. This could be problematic if there is a tight cooling-down time for the task, but subdivisions of tasks like DeliverDrink could relax the constraints by delaying the delivering of the drink if the cooling-down time is not reached.

For instance, in case that a VIP task appears just after a hot coffee is made and before delivering it, the scheduler has to consider all the cases explained in the Interrupting Tasks section. Basically, it tries to schedule everything by redoing the current subtask later (in the case of the drink in this example). If it cannot, it tries to redo the whole DeliverDrink task. If it is impossible, then it evaluates whether to cancel the current task or the new important task by using the gain measure. This means that the solver has to be called 1, 2 or 4 times, depending on the case.

Rescheduling can take some time, so although it can be done while the robot is traveling, opportunities may need a fast response to take advantage of them. In CoBot a reasonable time for each schedule could be 10 seconds, for example. With that threshold, we may need up to 40 seconds in the worst case scenario.

Experiments

The goal of this section is to evaluate the rescheduling architecture as an effective way to organize, execute and monitor the tasks performed by the robot in a social environment. Therefore, it is required to determine whether the system is

fast enough to provide good schedules in a small amount of time. In this sense, a maximum running time of 10 seconds (under regular computing capabilities) is desired as the maximum time that the robot should be waiting to the solver. This evaluation separately describes two main aspects of this work[1]. The first one is the performance of the developed scheduler in terms of solving time and schedule quality. The second one highlights the behavior of the Monitoring part by using an example task pool and three feasible schedules. Using robot execution time (for instance, navigation time) to reduce decision delays (using that time to compute new schedules or improve current ones) is out of the scope of this work.

MIP scheduler

The external MIP solver used is GLPSOL[2]. To cover the temporal requirement specified above, the solver permits to limit the time used to find a solution in several ways. The experiments performed are focused in two of them: limiting the maximum solving time and accepting a solution when a certain quality measure is reached. Solvers provide the relative MIP gap percentage, which gives an indirect insight of the quality of a solution. In essence, this measure is related to the current numerical upper and lower bounds, so when it reaches 0 % the solution is proven optimal. However, for clarity purposes, these experiments also use a function to represent quality (q) more directly:

$$q = (\sum_{i=1}^{n} e_i)/n$$

Quality is the sum of the end times for each task (e_i) divided by the number of tasks in the pool (n). The number by itself is meaningless, but when compared, it gives the average amount of time that a task will be delayed while using a configuration of the solver instead of another. It does not use priorities, but it is enough for the purposes of this section.

The experiments evaluate three configurations: 1) Solving time limited to 10 seconds with 4.4 % of relative MIP gap tolerance, 2) 10 seconds limit without tolerance and 3) 30 seconds limit. The memory usage is always under 100 MiB so there is no need to limit it too. The solver uses a base set of 480 random instances of task pools (set A). This set has some unfeasible or too hard instances so, to make some comparisons, the results also refer to subsets B and C which are contained in A. Subset B has 12 random instances per each size of the task pool, ranging from 1 to 15 (180 instances). They have been solved in all three configurations. Subset C is the same as B except that the ranges of pool sizes are limited from 8 to 15 (96 instances) to focus on the hardest ones.

The 4.4 % tolerance value used in the first configuration is the average relative MIP gap reached in the second configuration. Figure 2 uses the set B to show the average solving times for each task pool size and configuration, where instances have at least one solution in all cases.

[1]All experiments were carried out in an Intel Core i3-2330M CPU at 2.2 GHz with 4 GiB of RAM.

[2]http://www.gnu.org/software/glpk

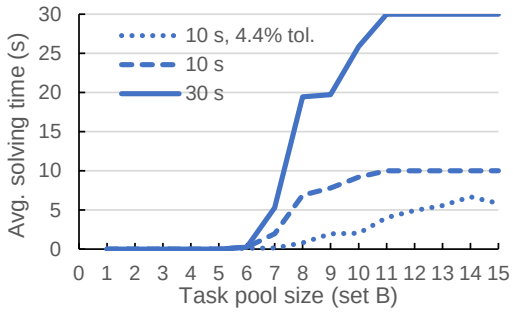

Figure 2: Evolution of the average solving time along different task pool sizes (set B ⊂ A).

The solver starts using the whole time allowed when the pool has 11 tasks in it, except when there is a MIP gap tolerance of 4.4 %. In this case, the average maximum solving time is under 8 seconds. That indicates that a valid solution is found fast and the rest of the time is used to optimize it or prove its optimality. The gap tolerance could be useful if the impact on the quality of the solution is acceptable for the intended application. Figure 3 shows the qualities for the set B. Time is measured in minutes.

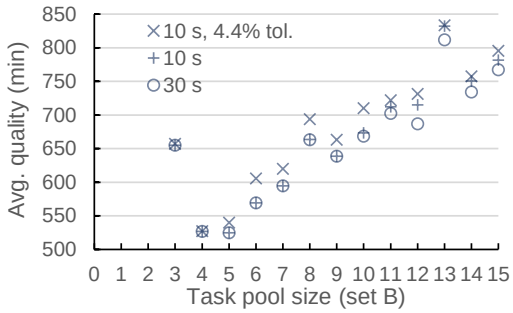

Figure 3: Average quality of the planned schedules with different pool sizes (set B ⊂ A). The Y axis starts at 500 to improve readability, but instances for sizes lower than 3 are hidden below.

Spending 30 seconds to plan the schedule is usually much better, in terms of quality, than using only 10 seconds with tolerance at 4.4 %. However, when there are 10 tasks or fewer, 10 seconds seem enough. CoBot should start responding in 10 seconds at most, although the results show that spending some seconds more can save much time. Table 2 details the 6 types of outcomes that the solver can return.

For the full set A of 480 instances, only 0.8 % of the instances passed the checks and were proven unfeasible. Also, only 11 % of the instances reached the time threshold without being solved. Sets B and C were solved by all configurations. Few optimal solutions were found from 10 to 30 seconds, but the average quality increased especially in the set C of difficult instances. Using the gap tolerance reduces the solving time sensibly but also the quality of the solutions. From 10 to 30 there is a reduction of 12 minutes of delay per task in set C. All these results suggest that the relative

MIP gap tolerance could be tuned to find a feasible schedule very fast to start responding as soon as possible, and then continue refining the schedule while the robot is working or waiting to start the next scheduled task.

	Configuration	10 s, 4.4% tol.	10 s	30 s
Set A	Time out: no solut.	11.0%	11.0%	8.8%
	Proven unfeasible	0.8%	0.8%	0.8%
	Check failed	4.4%	4.4%	4.4%
	Proven optimal	16.3%	42.7%	43.1%
	Min. gap reached	54.0%	0.0%	0.0%
	Time out: found	13.5%	41.0%	42.9%
	Solutions found	83.8%	83.8%	86.0%
Set B	Proven optimal	17.8%	51.1%	52.2%
	Min. gap reached	68.3%	0.0%	0.0%
	Time out: found	13.9%	48.9%	47.8%
	Av. solver time (s)	2.14 ± 3.6	5.07 ± 4.9	14.7 ± 14.8
	Av. quality (min)	611 ± 256	596 ± 250	590 ± 247
Set C	Proven optimal	0.0%	10.4%	12.5%
	Min. gap reached	74.0%	0.0%	0.0%
	Time out: found	26.0%	89.6%	87.5%
	Av. solver time (s)	3.98 ± 4.1	9.24 ± 2.5	26.88 ± 8.6
	Av. quality (min)	738 ± 135	721 ± 137	709 ± 137

Table 2: Solution type for the three configurations. Set C ⊂ B ⊂ A. Subsets B and C also have average solving times and qualities.

Monitoring

The target of this section is to evaluate how the rescheduling module works when some unexpected event occurs. Table 3 illustrates three possible schedules after the arrival of a VIP at minute 20. There are two kinds of tasks in each schedule: three *Deliver coffee* (C1, C2, C3) for the same person and one *Show off* (VIP). *Deliver coffee* is subdivided in two tasks that must be executed in order. All tasks have priority 1 except VIP, which has 100. The cooling down time for the coffee is 15 minutes. The time to travel from the initial location is 4 minutes. The time to travel from the coffee maker to the delivering location is 3. The VIP is in the same location of the coffee maker.

These schedules could be planned applying the described rescheduling policies in less than 10 seconds. This was because the pool is small enough and VIP can be scheduled before or after C1b, so there is no need to run the solver more times. Currently the system schedules from scratch when a new task arrives, so the results of the previous subsection are valid also in this one.

As it can be seen, the scheduler has locations into account, changing the duration of the tasks if the robot has to move to another place to perform them. Because of this, the scheduler tries to join all first parts and all second parts of *Deliver coffee* tasks to save trips. This is a clear improvement over the original scheduler. In schedule A, it is only possible to join two of them because the cooling down value of 15 minutes is too small.

Schedule A

Task	Start	End
…	0	10
C1a	11	20
C2a	21	26
C1b	27	31
C2b	32	33
C3a	34	42
C3b	43	47
VIP	48	53
Cost	739	

Schedule B

Task	Start	End
…	0	10
C1a	11	20
C2a	21	26
C1b	27	31
C2b	32	33
VIP	34	39
C3a	40	45
C3b	46	50
Cost	605	

Schedule C

Task	Start	End
…	0	10
C1a	11	20
VIP	21	23
C2a	24	29
C1b	30	34
C2b	35	36
C3a	37	45
C3b	46	50
Cost	454	

Table 3: Valid schedule examples after the arrival of a new task VIP at time 20. Units expressed in minutes.

The VIP task is performed at different positions in each table and this has an effect in the total time of the schedule and the cost that the scheduler has to minimize. Schedule A is the worst because the important task is executed later, and it takes 3 minutes more than the others because it needs an extra trip. Schedule B is much better because it saves one trip and executes the important task earlier. Schedule C executes the VIP task as soon as it arrives because the cooling down time is wide enough to perform VIP and C2a between C1a and C1b. All three schedules are valid because the only hard constraint is that VIP must start at or after minute 20. To execute the most important tasks first is a soft constraint.

As shown in the previous section of the experimentation, in scenarios with constrained task pools the optimal plan is not always obtained because of the need of fast responses. Sometimes false negatives occur (solutions not found by the solver that exist), which is undesirable especially when rescheduling a new task like VIP. However, in practice, the monitoring part works as intended because of the low planning time limit and because normally there are few remaining tasks in the task pool. There are also some online videos to watch the new scheduler at work[3].

Conclusion

This manuscript presents a new architecture of task execution, monitoring and rescheduling, which was developed in top of the current CoBots to improve their capabilities. Apart from the scheduling abilities of the original approach (Coltin, Veloso, and Ventura 2011), the new one is focused on fast solving times to ease the human-robot interaction, opportunity and failure detection, rescheduling with task interruptions, priority optimization and dependent tasks with cooling down times (a hot coffee cannot be delivered much later after being done). The developed component is generic enough to be applied in other multilevel architectures like NAOTherapist (González, Pulido, and Fernández 2017) by changing only the task specification and the communication interfaces. The experimentation shows that the system can perform well in normal conditions, although in scenarios with numerous remaining tasks the quality of the

[3] http://goo.gl/r2cszx

obtained schedules can be affected, especially if the allowed solving times are low.

Future work may include deeper integration with the current architecture to take control of some aspects that are managed in a lower level. The policies to take advantage of opportunities could also be improved by trying a quicker version of the VIP task, for example, in difficult task pools. This could lead in more reschedules, but they could be done while the robot is working. Also, alternative contingent plans could be considered, for example, to give a coke instead of a coffee because it is better than nothing. This would use more the concept of gain.

References

Alcázar, V.; Guzmán, C.; Prior, D.; Borrajo, D.; Castillo, L.; and Onaindia, E. 2010. PELEA: Planning, Learning and Execution Architecture. In *Proceedings of the 28th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG)*.

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; and Ridder, B. 2017. Opportunistic Planning in Autonomous Underwater Missions. *IEEE Transactions on Automation Science and Engineering (T-ASE)* PP(99):1–12.

Chen, D.-S.; Batson, R. G.; and Dang, Y. 2010. *Applied Integer Programming: Modeling and Solution*. John Wiley & Sons.

Coltin, B.; Veloso, M. M.; and Ventura, R. 2011. Dynamic user task scheduling for mobile robots. In *Automated Action Planning for Autonomous Mobile Robots, Papers from the 2011 AAAI Workshop, San Francisco, California, USA*.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory & Practice*. Elsevier.

Ghallab, M.; Nau, D.; and Traverso, P. 2014. The Actor's View of Automated Planning and Acting: A Position Paper. *Artificial Intelligence (AIJ)* 208:1–17.

González, J. C.; Pulido, J. C.; and Fernández, F. 2017. A three-layer planning architecture for the autonomous control of rehabilitation therapies based on social robots. *Cognitive Systems Research (CSR)* 43:232–249.

Guzmán, C.; Castejón, P.; Onaindia, E.; and Frank, J. 2015. Reactive execution for solving plan failures in planning control applications. *Integrated Computer-Aided Engineering (ICAE)* 22(4):343–360.

Rosenthal, S., and Veloso, M. 2012. Mobile Robot Planning to Seek Help with Spatially-Situated Tasks. In *Proceedings of AAAI'12, the Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Schermerhorn, P.; Benton, J.; Scheutz, M.; Talamadupula, K.; and Kambhampati, S. 2009. Finding and Exploiting Goal Opportunities in Real-Time During Plan Execution. In *Proceedings of the 21st International Conference on Intelligent Robots and Systems (IROS)*, 3912–3917.

Veloso, M.; Biswas, J.; Coltin, B.; and Rosenthal, S. 2015. CoBots: Robust Symbiotic Autonomous Mobile Service Robots. In *Proceedings of IJCAI'15, the International Joint Conference on Artificial Intelligence*.