An Integrated Control Framework for Long-Term Autonomy in Mobile Service Robots

Lenka Mudrova*, Bruno Lacerda* and Nick Hawes*.

***Abstract*— This paper describes an integrated framework for the long-term task-driven control of mobile service robots. The core components of the framework are: a high-level *task executor* that manages execution, for example by reacting to failures, or adding extra tasks required by the end-user on-the-fly; a *task scheduler* that schedules sets of tasks throughout the day, taking into account travel times between locations and task durations, while satisfying the time constraints associated with each task; and a *probabilistic topological motion planner* that provides time-dependent optimal navigation policies and expected navigation times between task locations. We illustrate the overall framework by reporting on a three-week deployment in a real-world office environment, and use the data collected during the deployment to validate and illustrate the capabilities of the framework to adapt itself to the different travel time expectations throughout the day.**

I. INTRODUCTION

Consider a mobile service robot operating in an office building for a long period of time, where it autonomously performs tasks to assist the occupants in their everyday activities. One can imagine a wide array of tasks for such a robot to execute, for example:

- *"Bring me a cup of coffee as soon as possible."*
- *"Check if there are people in office 123, between 20:00 - 20:15 today."*
- *"Check if the emergency exits are clear every 2 hours."*

Given that the robot is performing this type of tasks for long periods of time, they have different timing characteristics. For example, some are repetitive, being executed every day; others are requested by the end-user at arbitrary times, and the request might be for the robot to execute the task immediately, or execute it at a specific time in the future. Furthermore, the robot needs to navigate between different areas of the environment in order to arrive at its target location in time to perform the requested task. However, different areas of a building may present different navigation challenges at different times of day. For example, a cafeteria will be more crowded during lunch time, or office doors may be closed when the office is empty, or when there is a meeting. Therefore, the time the robot takes to navigate around a building is dependent on the time of day.

With this in mind, the autonomous execution of tasks in such office environments raises an array of interesting questions. These include:

1) When should task execution start in order to satisfy the given time constraints?
2) How much time does the robot need to travel between different task locations? How is it dependent on time of day?
3) How does the robot find robust and optimal navigation policies between different task locations?
4) How should the robot react if a new task is requested during the execution of another?
5) How can task execution be monitored, and what reactions are appropriate if a task fails or takes longer than expected to execute?

*School of Computer Science, University of Birmingham, UK. {lxm210, b.lacerda, n.a.hawes}@cs.bham.ac.uk
978-1-4673-9163-4/15/$31.00

Fig. 1: Our mobile service robot during deployment at G4S Technology.

In this paper, we leverage on our recent works on optimal high-level planning with probabilistic guarantees [1]; frequency-based learning and prediction of navigation durations and probabilities of navigation success [2]; and mobile robot task scheduling [3], to develop a control framework that deals with these issues in an integrated fashion. We evaluate our framework on data obtained from a three week deployment of a service robot in an office environment, at G4S Technology, Tewkesbury, UK (Fig. 1).

This deployment was based on the overall control architecture presented here, however some of its components were not integrated in the deployed system (e.g., the frequency based learning component). Thus, we will discuss the deployment details, along with the *routine* performed by the robot, and use the data gathered during the deployment to show how our approach provides a high-level controller that is able to *adapt* its task execution schedules to the expected environment dynamics at a given time of day.

II. RELATED WORK

The CoBot service robots [4] operate in an office building performing a variety of tasks. They use a *"User to Mobile*

Robot" architecture [5] running on a server. This system *manages incoming tasks* from a web-based user interface, *schedules* tasks across several robots [6], and keeps track of task *execution*. The robots autonomously navigate on a topological map [7], using Dijkstra's algorithm to find a path on the topological graph. Each robot performs given tasks, provides an on-board user interface and speech based *"interruptible autonomy"* [8] in order to modify, cancel or add a task. As of November 2014, the four CoBots have jointly travelled more than 1,000 km autonomously. A similar centralised system architecture is used by the mobile service robot Tangy [9] which performs a *sequence* of tasks rather than a *schedule*. A sequence differs from a schedule as exact start times for tasks are not specified, only their order.

In contrast to the previous architecture, robots Rin and Rout use a constraint network [10]. This network is continuously modified by an executor, a monitor and a planner in order to create configuration plans which specify causal, temporal, resources and information dependencies between individual actions. This general approach allows Rin and Rout to perform a variety of tasks.

Neither Tangy, nor Rin and Rout, are aimed at long-term behaviour. On the contrary, Willow Garage programmed a PR2 robot to perform an uninterrupted, 13 day run in an office environment [11]. The robot had only a simple executive framework which queued tasks for execution, but did use a novel failure recovery mechanism, including remote human teleoperation, to increase its robustness.

Finally, the CRAM software toolbox [12] provides a task executive that allows for the design, implementation, and deployment of autonomous robots that perform everyday manipulation activities, focussing on the integration of knowledge into the control process.

Contrary to our approach, none of the these works is able to cope with the set of questions we presented before in an integrated manner.

III. CONTROL FRAMEWORK

Our control framework, depicted in Fig. 2, runs on a single mobile robot and assumes the presence of navigation controllers which allows it to autonomously navigate on a topological map. We also assume the presence of other subsystems which are able to perform the tasks required of the robot (e.g. checking for the presence of people). A *task* is the main unit of behaviour within our cognitive control framework. It represents an instance of a behaviour that the robot should carry out. We refer to a single task by ω with numeric subscript i, for example ω_1. Its properties – listed below – are then referred by the same subscript.

- time properties:
  - a time window $\langle r_i, d_i \rangle$, where r_i is a *release date* (the earliest time instant when a task can start), and d_i is a *deadline* (the latest time instant when a task can finish);
  - a *processing time* p_i, which represents the expected duration of the task;

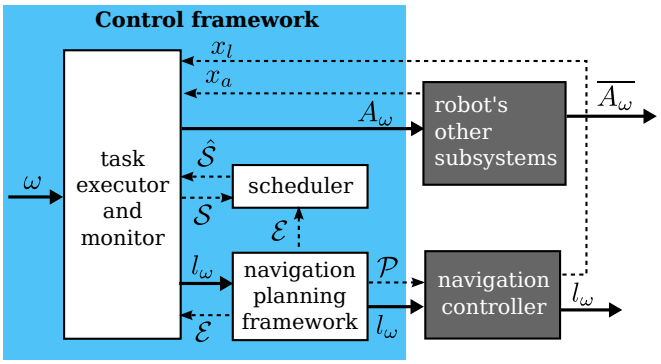

Fig. 2: An architecture of the proposed control framework, see Sec. III for an explanation.

  - a *start time of execution* s_i, to be defined by the scheduling procedure;
  - an *end time of execution* e_i, defined as $e_i = s_i + p_i$.
- a priority ψ_i;
- optional start l_i^s and end l_i^e locations;
- an *activity* to perform which has:
  - a name, for example "*check the fire extinguisher*";
  - a predefined sequence $A_{\omega_i} = (a_{i_1}, \ldots, a_{i_m})$ of actions how to fulfil the task. Actions are indivisible;
  - a boolean flag "interruptible" signalling if the task can be interrupted.

Our framework supports two types of task: *on-demand* tasks and *standard* tasks. A standard task should be added to the schedule and executed within its time constraints. An on-demand task should be executed immediately, thus the time window is not set for it. Tasks can be added to the control framework by humans via a web interface (e.g. requesting the robot to perform a task at a particular time); by components of the robot's other subsystems (e.g. requesting that a part of the map is explored at some time in the future); and by *routine scripts* which specify fixed sets of tasks for the robot to perform every day.

In order to address Question 4 (Q4), the executor maintains a set of tasks to be executed $\mathcal{S} = \{\omega_j, \ldots, \omega_l\}$ and it generates and executes a schedule $\hat{\mathcal{S}}$ for these tasks. The execution of an individual task is performed by a finite state machine which triggers navigation to the task's start location l_i^s and then execution of actions A_{ω_i}. It also uses internal feedback signals x_l, x_a from the navigation and action subsystems in order to react to failures (Q5).

Our framework is focused on a single robot and, based on this, we assume that tasks cannot be interleaved or performed in parallel. The robot is therefore the only resource for its scheduler to manage (Q1). The scheduler's input is the set $\mathcal{S}$ of tasks, and the output is a schedule $\hat{\mathcal{S}}$, which is an ordering of $\mathcal{S}$, along with the corresponding start times s_i and end times e_i for each task $\omega_i \in \mathcal{S}$. The scheduler creates $\hat{\mathcal{S}}$ such that the overall waiting time for task execution is minimised. Moreover, a valid schedule ensures that a robot has enough time to travel between the end location l_i^e of each finished task and the start location l_i^s of the

subsequent task. This is done by considering *travel duration estimations* $\mathcal{E}$ between the end and start locations of all pairs of tasks. Our framework provides an optimal topological navigation framework (Section III-C) which acts both as a travel time estimator for the scheduler, and a topological motion planner that is called by the executor to generate and execute optimal navigation policies $\mathcal{P}$. This navigation framework uses statistics gathered from long-term experience to adapt $\mathcal{P}$ to the dynamics of the environment, addressing (Q3, Q2).

While the robot executes a task ω_i, the task monitor observes how much time the actions have already consumed and compares it to the expected end of execution e_i (Q5). If the execution of a task does not end before reaching this processing time p_i, the monitor sends an interruption signal to the overrunning task, causing its cancellation. The same monitoring and interruption procedure is also done for navigation to the task location l_i^s, with the time taken to navigate being compared with the estimated travel time $\mathcal{E}$.

A. Task Executor

The task executor is the main control component in our framework. It manages task execution and scheduling, whilst also reacting to various forms of failures. When a new standard task is added to the executor, it updates its schedule using the *trySchedule()* method (described below) and continues with task execution as described by its current schedule ($\hat{\mathcal{S}}$). When an on-demand task is added it cancels the currently executing task (if it is interruptible) and executes the new task instead, whilst rescheduling its remaining tasks, taking the new situation into account.

The method *trySchedule()* (Alg. 1) calls the scheduler, which tries to find a schedule for a set containing old tasks $\mathcal{S}_o$ (previously scheduled) and newly added tasks $\mathcal{S}_n$. If it does not succeed, method *drop($\mathcal{S}_n$, 0.2)* (Alg. 2), drops 20% of the new tasks ($\mathcal{S}_d$) and overwrites $\mathcal{S}_n$ with the remaining 80%. The dropped tasks are saved because the robot might still be able to schedule some of them in the future.

Algorithm 1 trySchedule($\mathcal{S}_n$)

```
while not scheduler({S_o, S_n}, Ŝ) and S_n ≠ ∅ do
  S_n, S_d, preemptionNeeded = drop(S_n, 0.2)
  if preemptionNeeded then
    S_n = S_o + S_sn
    startExecution()
  end if
end while
return Ŝ
```

The method *drop($\mathcal{S}_n$, $amount$)* (Alg. 2) uses task priorities ψ_{ω_i} in order to drop less important tasks if a schedule cannot be found. The method *theLowestPriorityTasks($\mathcal{S}_n$,amount)* creates a subset $\mathcal{S}_d$ of tasks with the lowest priority in set $\mathcal{S}_n$. If this subset contains more than $amount \cdot size(\mathcal{S}_n)$ tasks, $\mathcal{S}_d$ is reduced to that amount, by randomly choosing tasks in $\mathcal{S}_d$ and moving them back to $\mathcal{S}_n$. If there is no task currently executing, tasks $\mathcal{S}_d$ are dropped. However, if there is a task executing, the executor checks whether the tasks to be dropped have a higher priority than the executing task. If they do, then execution must be interrupted so that the higher priority tasks can can propagated to execution. If the executing task cannot be interrupted, the framework is forced to drop the higher priority tasks.

Algorithm 2 drop($\mathcal{S}_n$,amount)

```
S_d = theLowestPriorityTasks(S_n,amount)
if not taskCurrentlyExecuting then
  return S_n \ S_d, S_d, false
else
  if smallestPriority(S_d) ≤ taskExecuting.priority then
    return S_n \ S_d, S_d, false
  else
    if isCurrentTaskInterruptible then
      return S_n, ∅, true
    else
      return S_n \ S_d, S_d, false
    end if
  end if
end if
```

1) Execution control and monitoring: The task executor's behaviour is driven by the schedule. Execution proceeds sequentially through the schedule, taking the next task to be executed, triggering navigation to the start location of the task, then triggering action execution. Before starting navigation, the executor checks whether it is the correct time to execute the task, i.e. that the estimated travel duration will cause the robot to arrive at the start location after release date r_i, in time to complete the task before deadline d_i. If it is too early for this, then the executor simply causes the robot to wait at its current location for the minimum duration before the r_i constraint can be satisfied. If the estimated arrival time plus execution time will exceed d_i then the task is considered failed, and rescheduling is performed with the remaining tasks.

While the robot is navigating to the start location of a task, the task executor monitors both the duration of travel, and whether it arrives successfully. If the duration exceeds the estimated duration $\mathcal{E}$ by a configurable proportion then navigation is preempted. If this happens, or navigation itself reports failure, then the task is cancelled and considered failed. Similar monitoring is performed during task execution, with it being preempted if it exceeds its stated processing time p_i by a configurable proportion.

Tasks can provide a feedback signal stating whether they are interruptible or not. This is used to prevent the task executor from preempting an action sequence that should not be interrupted, e.g. when the robot is interacting with a human, or performing some crucial processing. This flag can change its value at runtime and is checked before any of the above preemptions are triggered.

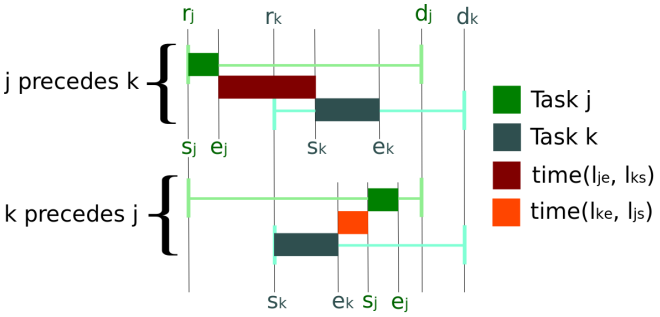

Fig. 3: Visualisation of two possible situations for task ω_j and ω_k, which has overlapping time windows. Tasks are same in both situations, but the travel time differs.

B. Scheduler

Schedules are created using *Mixed Integer Programming* (MIP), using the tasks properties to create a set of constraints. We then use the SCIP solver [13] to find a solution to the set of constraints with the following optimisation criterion:

$$\min \sum_{i \in \{1...m\}} e_i \tag{1}$$

The first constraint ensures that a task ω_j is executed within its time window:

$$r_j \leq s_j \ \wedge \ e_j \leq d_j. \tag{2}$$

The second constraint makes sure that execution of tasks does not overlap. Therefore, two tasks ω_j, ω_k are performed in one of the following orderings: O1: $\omega_j < \omega_k$ (read ω_j precedes ω_k) or O2: $\omega_k < \omega_j$, see Fig. 3. Between execution of task ω_j and ω_k, a robot must travel between locations l_j^e and l_k^s and it consumes $\mathcal{E}(l_j^e, l_k^s)$ time units. Notice, that $\mathcal{E}(l_j^e, l_k^s) \neq \mathcal{E}(l_k^e, l_j^s)$ in a general case as illustrated in Fig. 3. This is represented by the following constraint:

$$s_j + p_j + \mathcal{E}(l_j^e, l_k^s) - s_k \leq 0. \tag{3}$$

$$\vee$$

$$s_k + p_k + \mathcal{E}(l_k^e, l_j^s) - s_j \leq 0. \tag{4}$$

Solving scheduling problems using MIP is a standard technique in the community [5]. In [3], we demonstrated how it is possible to simplify the MIP problem by choosing between Eq. (3) or Eq. (4) when constructing the constraints. This allows our framework to schedule hundreds of tasks at a time with only a minor reduction in the optimisation criteria with respect to optimal.

C. Topological Navigation with Probabilistic Guarantees

In order to have robust navigation in populated, dynamic, environments, lower level motion controllers must be integrated with higher level discrete planners. Ideally, the higher-level planner should also be able to provide expectations on travel times, and use the experience gathered by the robot to improve such expectations. These expectations can then be provided as an input to the scheduling mechanism.

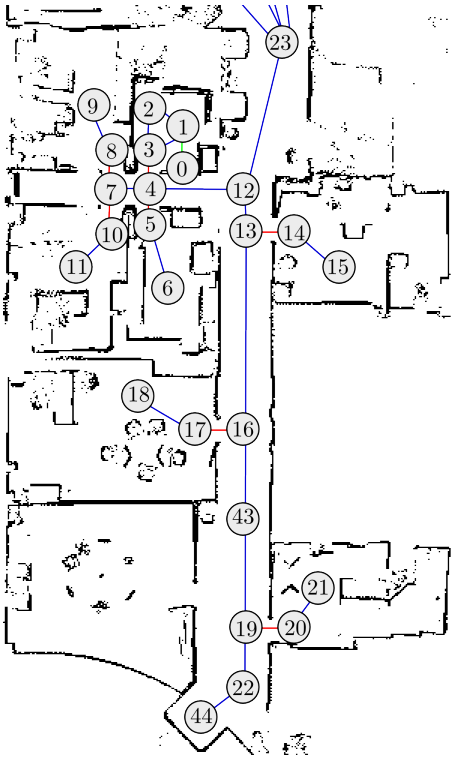

Fig. 4: A fragment of G4S metric and topological maps. Blue edges use default ROS navigation, red edges use a specialised door crossing behaviour, and the green edge uses a docking behaviour. All edges are bi-directional.

Thus, in our approach, the high-level motion planner is used both for generation and execution of optimal plans, and for calculation of expected travel times that are used to inform the scheduler, providing it with accurate estimations for different times of day.

Our approach is based on building time-indexed *Markov decision process* (MDP) models of topological maps. We define a topological map as a set of k topological nodes $V = \{0, ..., k-1\}$, representing locations in the environment, and a set of topological edges $E \in V \times V$, providing mappings that represent the motion controller to be used to navigate between nodes. A fragment of the topological map used in the G4S deployment is depicted in Fig. 4.

We developed a *topological navigation executor* that executes and monitors the navigation between topological nodes, using the corresponding motion controller, and logs data on this execution to a database. This data is then used to build *frequency models* for each edge (see [2] for details). Such models provide, for each edge e, time-dependent probabilities of success $p_e(t)$, and execution time expectations $\tau_e(t)$. They can then be used to build time-indexed *Topological Map MDPs*: For a given time $t \in \mathbb{R}_{\geq 0}$, we define $\mathcal{M}_t = \langle S, \bar{s}, A, \delta_t, c_t \rangle$, where: (i) $S = V$ is a finite set of states, corresponding to the topological nodes; (ii) $\bar{s} \in S$ is the initial state, corresponding to the initial position of the robot in the environment; (iii) $A = E$ is a finite set of

actions, corresponding to the edges in the topological map; (iv) $\delta_t : S \times A \times S \to [0,1]$ is a probabilistic transition function, where $\sum_{s' \in S} \delta_t(s,a,s') \in \{0,1\}$ for all $s \in S$, $a \in A$. For $i,j \in S$, if there is an edge $e = (i,j) \in E$, we define $\delta_t(i,e,j) = p_e(t)$, $\delta_t(i,e,i) = 1 - p_e(t)$ and $\delta_t(i,e,j) = 0$ for all $j \in S \backslash \{i,j\}$; and (v) $c_t : S \times A \to \mathbb{R}_{\geq 0}$ is a cost function, representing the expected edge transversal time at time t. For $i,j \in S$, if there is an edge $e = (i,j) \in E$, we define $c_t(i,e) = \tau_e(t)$.

Then, we use the work in [1] to generate a *cost-optimal policy*, which, for *all* topological nodes $v \in V$, maps v to the optimal edge to be transversed to reach a target node v_t, and provides an expected travel time from v to v_t.

Note that, in our integrated framework, we are only using *single state reachability* specifications. However, [1] allows for the specification of more general tasks, expressed in co-safe linear temporal logic. Extending our integrated control framework to allow for this more general class of specifications is subject of future work.

As we will see in the next section, our current approach provides good travel time estimations, and robust optimal policies. These allow our integrated framework to efficiently adapt its behaviours, taking into account the predictions on navigation times for different times of day.

IV. EVALUATION

In this evaluation, we use a dataset gathered in 2014, during a three-week long deployment of our robot in the offices of G4S Technology, Tewkesbury, UK. This deployment environment has approximately 300 m^2 and is used by approximately 20 people. Every week day between 22/5/14 and 13/6/14, the robot followed a routine – using the control framework presented in this work, without the travel times learning component – where it would perform a range of tasks (checking fire doors and fire extinguishers, building 3D maps, searching for objects) in preconfigured time windows (roughly three hour slots from 08:45 onwards), returning to a docking station when idle. During this time, the robot covered 20.64km and completed 963 tasks successfully. Fig. 5 shows how the tasks were executed throughout the deployment.

To illustrate and evaluate the integration of our scheduling mechanism with the topological motion planner, and its ability to adapt to the learned travel times, we split the dataset into training data (the first two weeks of the deployment), and testing data (the last week of the deployment).

A. Influence of environment dynamics

To evaluate the long-term adaptation of our framework we compare the following approaches for modelling navigation in a dynamic environment:

- $W0$: MDP using distance-based estimates, no data;
- $W1$: MDP model using data from just the first week;
- $W2$: MDP model using data from two weeks.

First, the 435 paths which the robot travelled during the testing week are obtained from the dataset. For each path we obtain the ground truth value g which states how long

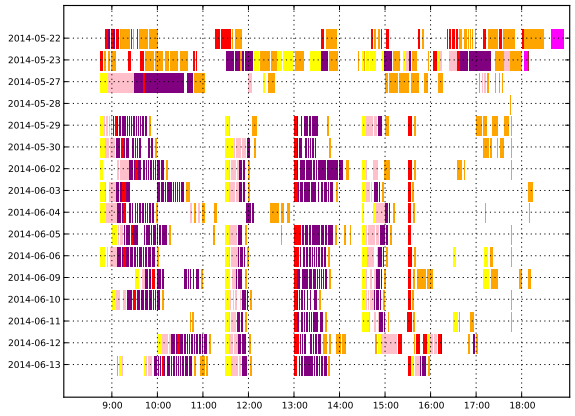

Fig. 5: The tasks performed by the robot. Key: yellow, object checks; red, door checks; purple, 3D mapping; pink, object search; orange, waiting.

the navigation actually took. Then, a set of time estimates $\mathcal{E}_X$ on each path is obtained from the various MDP models, where $X = \{W0, W1, W2\}$.

1) Methodology: The relative difference between an estimate and the ground truth is calculated as follows:

$$\Delta\mathcal{E}_X = \frac{\mathcal{E}_X - g}{g}. \tag{5}$$

Positive values of $\Delta\mathcal{E}_X$ mean that the particular method overestimates the time needed to travel between tasks. In terms of robot behaviour this typically means the robot will arrive early for tasks and must therefore wait around (wasting time) before executing the associated actions. In contrast, negative values mean that a schedule is created that underestimates the time needed to travel between task locations. As a result, the robot will fail to successfully complete the schedule. Therefore underestimates are a more damaging form of estimation error in our framework.

2) Results: The absolute mean errors and corresponding standard deviations of the relative results, plus their quantiles, on the 435 paths are reported in Tab. I. Estimates for selected paths from Fig. 4 are visualised in Fig. 6. It can be observed that the MDP with the most training data, $\Delta\mathcal{E}_{W2}$, provides the best estimates (lowest mean error). As would be expected, the non-adaptive model based during on distance $\Delta\mathcal{E}_{W0}$ underestimates by a larger degree to either adaptive model. After one week of data the adaptive model ($\Delta\mathcal{E}_{W1}$) substantially overestimates most of the paths. When a second week of data is added, more experience of environment causes the estimates to increase in accuracy. $\Delta\mathcal{E}_{W2}$ still overestimates but significantly less than model $\Delta\mathcal{E}_{W1}$. As can be observed in Fig. 5, during the first days of the deployment the routine was not followed very well (due to bugs in other parts of the system). Due to this, the data available for training $\Delta\mathcal{E}_{W1}$ has a different temporal distribution to the testing data. The second week of data corrects this, demonstrating that *long-term* experience improves our approach.

set	0.1-q	0.25-q	0.75-q	0.9-q	error	std-dev
$\Delta\mathcal{E}_{W0}$	-0.457	0.141	1.844	2.714	1.155	0.895
$\Delta\mathcal{E}_{W1}$	-0.151	0.244	2.088	12.823	8.522	30.526
$\Delta\mathcal{E}_{W2}$	-0.189	0.227	1.171	1.896	0.982	1.082

TABLE I: The results of travel time estimates on the dataset.

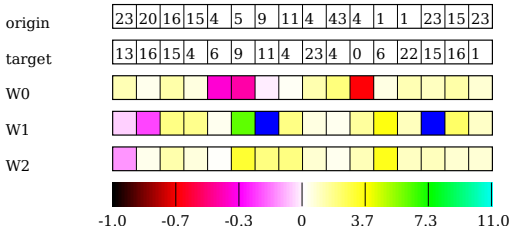

Fig. 6: Selected estimated relative travel times.

B. Adaptive Scheduling using Travel Time Estimations

The time-dependent travel time estimations also have an influence on the scheduling mechanism itself, as it tries to build a schedule that does as many tasks as soon as possible. To illustrate this, we generated a test scheduling problem, comprised of 16 tasks across 13 different locations. Fig. 7, depicts the schedules computed for different times of day.

It can be observed that the set of tasks can be scheduled more compactly as the expected travel times get less over-estimated as a result of the larger amount of data used for learning. Furthermore, the results with one week of learning data illustrate the adaptive nature of our framework: the schedules are different throughout the day. However, this variation after one week of learning is largely incorrect. Because our environment was not very dynamic after two weeks of data the schedules look the same throughout the day. Notice though that they are different to the schedules based on distance estimates, as our model adapts to the robot's behaviour in the environment. We are in the process of a new deployment in a more dynamic office environment, where we expect to have adaptive schedules even after longer periods of learning.

V. CONCLUSION

We presented an integrated framework for long term deployments of mobile service robots in office environments. We illustrated how our approach allows a service robot to robustly execute a variety of tasks, with different timing requirements, in such an environment. Our framework is implemented as a set of ROS packages, available online at `https://github.com/strands-project/strands_executive`.

Future work includes further evaluation of our approach in real-life scenarios; extending the topological motion planner to include our recent developments on policy generation for MDPs [14]; and extending the scheduler such that it creates schedules with improved robustness, by using the probabilities of navigation satisfaction that can be provided after integrating [14] in the control framework.

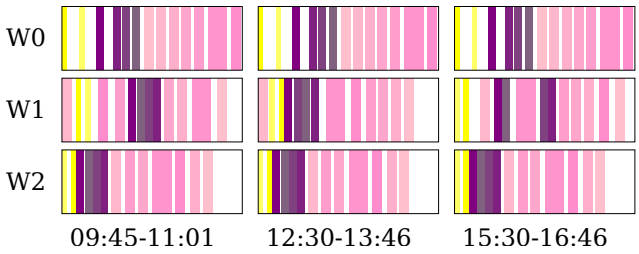

Fig. 7: Schedules using different methods for travel time estimates. White space between tasks represent the expected time to travel between tasks locations.

ACKNOWLEDGMENT

The research leading to these results has received funding from EU FP7 under grant agreement No 600623, STRANDS.

REFERENCES

[1] B. Lacerda, D. Parker, and N. Hawes, "Optimal and dynamic planning for Markov decision processes with co-safe LTL specifications," in *Proc. of 2014 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2014.

[2] J. P. Fentanes, B. Lacerda, T. Krajník, N. Hawes, and M. Hanheide, "Now or later? Predicting and maximising success of navigation actions from long-term experience," in *Proc. of 2015 IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2015.

[3] L. Mudrova and N. Hawes, "Task scheduling for mobile robots using interval algebra," in *Proc. of 2015 IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2015.

[4] M. M. Veloso, J. Biswas, B. Coltin, S. Rosenthal, T. Kollar, C. Mericli, M. Samadi, S. Brandao, and R. Ventura, "Cobots: Collaborative robots servicing multi-floor buildings." in *Proc. of 2012 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2012.

[5] B. Coltin, M. M. Veloso, and R. Ventura, "Dynamic user task scheduling for mobile robots." in *Proc. of 2011 AAAI Workshop on Automated Action Planning for Autonomous Mobile Robots*, 2011.

[6] B. Coltin and M. M. Veloso, "Online pickup and delivery planning with transfers for mobile robots," in *Proc. of 2014 IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2014.

[7] J. Biswas and M. M. Veloso, "Localization and navigation of the cobots over long-term deployments," *The International Journal of Robotics Research*, vol. 32, no. 14, 2013.

[8] B. C. Yichao Sun and M. Veloso, "Interruptible autonomy: Towards dialog-based robot task management," in *Proc. of 2013 AAAI Workshop on Intelligent Robotic Systems*, 2013.

[9] W.-Y. G. Louie, T. S. Vaquero, G. Nejat, and J. C. Beck, "An autonomous assistive robot for planning, scheduling and facilitating multi-user activities," in *Proc. of 2014 IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2014.

[10] M. D. Rocco, F. Pecora, and A. Saffiotti, "When robots are late: Configuration planning for multiple robots with dynamic goals." in *Proc. of 2013 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2013.

[11] W. Meeussen, E. Marder-Eppstein, K. Watts, and B. P. Gerkey, "Long term autonomy in office environments," in *ICRA 2011 Workshop on Long-term Autonomy*, 2011.

[12] M. Beetz, L. Mösenlechner, and M. Tenorth, "CRAM – A cognitive robot abstract machine for everyday manipulation in human environments," in *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2010.

[13] T. Achterberg, "SCIP: Solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, no. 1, 2009.

[14] B. Lacerda, D. Parker, and N. Hawes, "Optimal policy generation for partially satisfiable co-safe LTL specifications," in *Proc. of 24th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2015.