Computational Resource Management of Multi-Channel Controller

Hodai Goldman

October 25, 2017

Contents

1 Introduction **3**

2 Problem Statement **4**

3 The proposed approach **5**

3.1 Architecture 6

3.2 Guarded Automata as Interfaces for Control and Scheduling 6

3.3 Implementation in an Auto Pilot Software 7

3.4 Integration With a Kalman Filter 7

4 Architecture: Automata Base Scheduler (From proposal) **8**

4.1 Sensors 9

4.2 State Estimator 9

4.3 Control Tasks 9

4.4 Computation Resource Scheduler: Automata Based Scheduler 10

4.4.1 The Proposed Scheduling Methodology 10

5 A demonstration of the approach in simulation **12**

6 Case Study: Stabilizing a quadrotor in front of a window **14**

6.1 Observer Design 15

6.2 Controller Design 16

6.3 Analysis and Specification Automata 17

7 Results **18**

8 Scheduler Prototype **22**

8.1 Formal Definition of Guard (specification) automata 22

8.2 Automata Operation 22

8.3 GOAL Tool 22

9 Testing Environment **22**

9.1 Experiment setup (From Paper) 22

9.2 APM Controlled Quad-copter 23

9.3 Vision based Autonomous In-Door Flying 23

9.4 Reference Measurement with OptiTrack System 24

10 Related Work **24**

11 Conclusions and Future Work **24**

12 Apendix **24**
12.1 Kalman Filter and State Estimation 24
12.2 Raspberry-Pi Camera Driver 26
12.3 TODO 26
12.4 TODO 26

1 Introduction

Cyber-physical systems (CPS) technologies of integrating software and control are at the heart of many critical applications (see [19] for a review of such applications). These technologies aim at handling issues that emerge when the interaction of software and hardware brakes the traditional abstraction layers: when researchers and practitioners are required to consider a unified view that includes both software and hardware. An example of such an issue is the challenge of dynamic assignment of computational resources to software based controllers discussed in, e.g., [3, 28, 30]. While the computation burden required by the control loops can be ignored in many situations, this is not always the case. A main motivating example studied in this thesis is vision based control, where computer vision algorithms acquire state information to be used in a feedback loop (see, e.g., [9, 13, 26]). Unlike conventional sensors such as accelerometers, gyros, compasses, etc., a visual sensor requires significant processing of the acquired image to prepare the state information for feedback. Since typical cyber-physical application, such as robot control, consist of many control loops, responsible for different aspects of the system, that run simultaneously and share the same computational resources, computer vision algorithms cannot always be invoked in full power. Alternatively, we propose in this thesis a mechanism to dynamically trade CPU consumption vs. measurement accuracy so that data acquisition algorithms run in full power only when the control loop requires accurate data.

A main challenge in forming mechanisms for the integration of software and control lies in the design of efficient interfaces for integrating the engineering diciplines involved (see, e.g., [30]). Components with clearly specified APIs, such as Java library classes, allow designers to build complex systems effectively in many application domains. The key to such modular development is that each component can be designed, analyzed, and tested without the knowledge of other components or the underlying computing platform. When the system contains components with real-time requirements, the notion of an interface must include the requirements regarding resources, and existing programming languages provide little support for this. Consequently, current development of real-time embedded software requires significant low-level manual effort for debugging and component assembly (cf. [15, 18, 25]). This has motivated researchers to develop compositional approaches and interface notions for real-time scheduling (see, e.g., [5, 8, 10, 21, 23, 24, 27, 29]).

In this thesis we present an approach, a proof-of-concept implementation, and a case study in scheduling computations in embedded control systems. The proposed design is based on the automata based scheduling approach, suggested in [1, 2, 14, 20, 30], where automata are proposed as interfaces that allow the dynamicity and efficiency of desktop operating systems with the predictability of real-time operating systems. The approach allows for components to specify the CPU resources that they need in a way that gives an application agnostic scheduler the freedom to choose schedules at run-time such that the

needs of all the components are taken into account, even of components that were added only at run-time. The main contributions of this thesis relative to the earlier work in this direction is: (1) We propose an extension of the automata based scheduling framework that allows to direct the schedule based on the state of the controllers; (2) We propose a technique, based on the theory of Kalman Filters, for designing reactively scheduled controllers; (3) We report on our experience with improving the performance of a real-time, vision-based, control system (a drone that stabilizes itself in front of a window).

2 Problem Statement

During this thesis we are concentrate on feedback control loop based systems as shown in Figure 1. We assume a close (feedback) control loop where the physical plant, *System* state x, is monitored via an array of sensors (*Sensing*) which produce measurements array y that represents noisy sample of some functions of the state variables. We assume uncertainty observations from the sensors so, after sensing, an entity called *State Estimator* aggregates the measurements from all the sensors (y array) and makes an educated estimation ($\hat{x}$) of the current state. Then the *Control Law* entity generates the controller outputs (u), control the actuators, to change the state of the system in order close the gap between the current estimated state and the reference desire state (*Input* r).

In this thesis, for analysis, we assume a general *linear dynamical system* discretized in the time domain. At each discrete time increment, the true state x_k is evolved from the previews state x_{k-1} according to:

$$x_k = Ax_{k-1} + Bu_k + w_k$$

Where, A is the state transition model which is applied to the previous state x_{k1}, B is the control-input model which is applied to the control vector u_k, and w_k is the process noise which is assumed to be drawn from a zero mean normal distribution with covariance Q_k. At time k an observation (or measurement) y_k of the true state x_k is made according to:

$$y_k = Cx_k + v_k$$

Where, C is the observation model which maps the true state space into the observed space and v_k is the observation noise which is assumed to be zero mean Gaussian white noise with covariance R_k.

The current state-of-the-art cyber-physical control systems, as descibed e.g. in [**?**], are developed as follows, control engineers first design control tasks (control and estimation tasks) as periodic computations, then they specify the required periodic frequency for the task, and then software engineers design a scheduler that ensures the periodic frequency requirements are met. The last step is usually done using pre-computed knowledge of

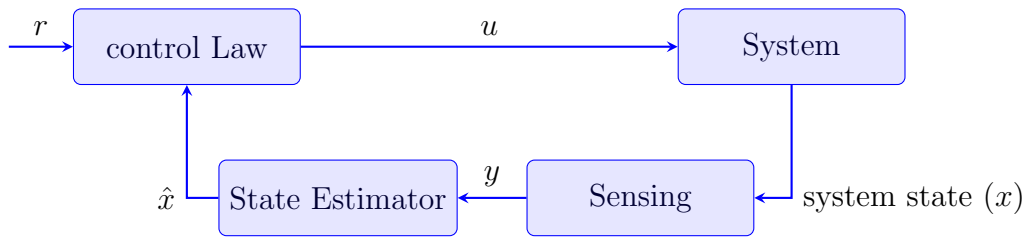


Figure 1: A typical feedback control loop where the physical plant (*System*) is monitored via an array of sensors (*Sensing*) which produce noisy sample of the state variables y. And after sensing, *State Estimator* aggregates the measurements from all the sensors and produce estimation of the current state $\hat{x}$. Then the *Control Law* produce output to the actuators (u) in order to close the gap between the current estimated state and the reference state (r).

the expected (maximum) duration of the tasks. Note that in order to ensure that the controlled plant is always properly maintained, for example engine temperature never exceeds maximal safe temperature, the periodic frequency must be tuned for the worst-case state that the system might be in.

We show how to achieve better performance and better resource utilization for control systems by using richer and more flexible requirements model for the tasks. Specifically, we develop tools and methodologies such that control engineers are able to specify more accurately the requirement features of their control tasks, that the scheduler will use for executing dynamic resource assignment that will, at the same time, guarantee required control performance and will be efficient in its use of computational resources.

3 The proposed approach

As proposed in earlier work on automata based scheduling ([1, 2, 30]) we aim at a development process where a system is built as a composition of a set of components where each component is a software module (a set of procedures) accompanied with an automaton. Our addition here is that we allow the automata to be guarded, i.e., each automaton acts as a specification of a reactive system that tells the scheduler which functions of a component it may run in each slot depending on the **dynamic state** of the controllers. A second addition is that we have implemented the approach as an enhancement of the internal scheduler of the ArduPilot Mega (APM) [11] open source unmanned vehicle autopilot software suite. A third contribution of this thesis is a proposal of a specific way to use the automata based scheduling framework with a Kalman filter. We elaborate on each of these in the following subsections:

3.1 Architecture

In our methodology, we use rich component specification represented by automata. As presented above, the first advantage of this technique is the ability of the scheduler to "react" the state of the system, therefore we need more intense interaction between the *Scheduler* and the control loops in particularly with the estimator. The architecture of control system as we believe, is illustrated in Figure 2. The architecture, that is based on modern controller architecture where the system has multiple controlling tasks, comes to support an efficient scheduling protocols in modern control systems consisting of a processor that runs all the tasks of many independent control loops in the system. Each control loop (blue components in Figure 2) will tell the *scheduler* of its level of certainty ($P = var(\hat{x} - x)$) and the scheduler will allocate resource (processor time) based on this data, meaning the *scheduler* allocate resource dynamically based on the system current needs rather than the worst case needs, making the scheduler an active part of the control loop.

Our methodology is general and may be applicable in a wide range of applications. However, in this initial phase of the research, we focus on a specific sub-domain and in handling all technical issues in order to prove the concept.

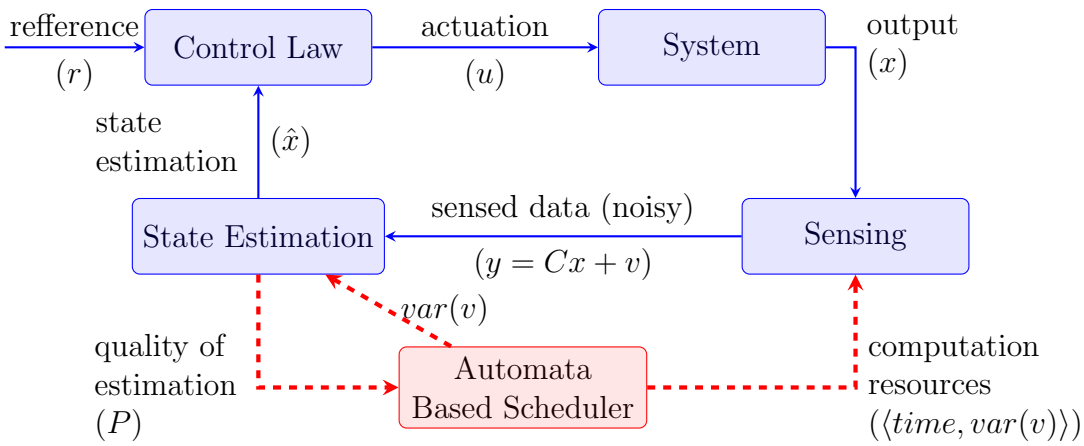


Figure 2: The controller framework we implement, Each control loop (depicted in blue) informs the resource allocator (Scheduler) of its quality of estimation, the *scheduler* will allocate CPU time accordingly in order to maintain valid estimation quality. Specifically for the Sensing processes, for example in our case *Computer Vision* task, the guard automaton specify the required sensing quality ($var(v)$) which determines the required execution time ($time$). The Sensing alocation is noted by $\langle time, var(v)\rangle$.

3.2 Guarded Automata as Interfaces for Control and Scheduling

As our goal is to allow dynamic selection of the computation load in the feedback loops based on the states of the systems, we start with a general software architecture in which

each component (implementation of a specific feedback loop) is represented by a code module (in our case, a class in C++) and an automaton that specifies when to invoke its methods. The transition relation of the automaton depends, in addition to the current state, also on data produced by the estimator of the feedback loop (we experimented with different options for this data, as discussed below).

The motivations of using automata as described above are: (1) Automata allow for a rich specification language; (2) It is easy to construct schedules that obey the specification with negligible computational burden; (3) Automata theory gives a solid framework for composing the specifications of competing requirements for analysis and for schedule synthesis.

In this thesis we focus on the first two motivations in the above list. The third is discussed in details in earlier works on automata based scheduling and is the focus of future paper that we are preparing where we describe some analysis techniques we have developed for guarded automata.

3.3 Implementation in an Auto Pilot Software

As our main case study is in flight control (see Section 6), we chose the ArduPilot Mega (APM) platform for experiments [11]. To this end, we implemented a basic automata based scheduler for this platform. The built-in task scheduling specification in APM consist of a table as shown in Figure 3. This table is easy to maintain and to adjust, but it is used under an assumption that there is enough CPU power to run all the tasks in the specified frequencies. APM does contain a mechanism to handle overruns, by moving tasks to the next window when there is not enough time to run them now, but the system is designed under the assumption that this only happen in rare situations.

To allow a reactive scheduler that runs different modes of the software-based controllers depending on their state, we replaced the scheduling table in our version of the APM with automata that specify when to run the tasks. Note that automata allow for specifying the requirements that the table represents, using simple circular automata without guards (see, e.g., [30]). Automata, however, can model more advanced scheduling instructions with very little addition to the complexity of the scheduler, as we will demonstrate in Section 5 and in Section 6 below.

3.4 Integration With a Kalman Filter

The third layer of the approach we propose is based on the observation that a standard Kalman filter produces information that can be used to guide the automata of the components.

As we will elaborate in the description of the simulations and of the case study below, we propose to schedule the functions that implement algorithms for sensing and for actu-

```
/*
scheduler table - all regular tasks apart from
the fast_loop() should be listed here, along
with how often they should be called (in 10ms
units) and the maximum time they are expected
to take (in microseconds)
*/
static const AP_Scheduler::Task
scheduler_tasks[] PROGMEM = {
{ update_GPS,            2,     900 },
{ update_nav_mode,       1,     400 },
{ medium_loop,           2,     700 },
{ update_altitude,      10,    1000 },
{ fifty_hz_loop,         2,     950 },
{ run_nav_updates,      10,     800 },
{ slow_loop,            10,     500 },
{ gcs_check_input,       2,     700 },
{ gcs_send_heartbeat,  100,     700 },
{ gcs_data_stream_send,  2,    1500 },
{ gcs_send_deferred,     2,    1200 },
{ compass_accumulate,    2,     700 },
{ barometer_accumulate,  2,     900 },
{ super_slow_loop,     100,    1100 },
{ perf_update,        1000,     500 }
};
```

Figure 3: APM scheduling specification

ation based on the error variance of the estimation that a Kalman filter provides, the idea is to operate different sensing modes in order to regulate the estimation error variance, see Section 12.1 for Kalman-filter characteristics.

4 Architecture: Automata Base Scheduler (From proposal)

As describe in Section 2 control systems have two main parts: *state estimation* and *controller*. The common used techniques for controllers, e.g PID, are relatively low resources consumers and there is no real justification of optimizing it, but this is not always the case for estimation process, some times there are heavy computational tasks as part of state estimation task, this refers to both the estimator and sensing stage. The sensor can be complex computations of Global Positioning System (GPS) in small processor or even camera with heavy computer vision. Those sensing task need to run in real-time and can interfere with rest of the system tasks. In our architecture we concentrate more in those heavy sensing processes in order to improve the resource utilization and control performance. (See **??**)

TODO - need to fix here!!!

Below we will dive in each part of the new architecture and explain how it should be adjusted.

4.1 Sensors

The sensing process are assumed to be periodic task. In order to allow dynamic scheduling it must be able to execute in a variable period between executions or to allow control of the execution duration time. There are two general approaches we consider for this task: use *any-time* algorithms or finite *execution-modes* based tasks. *any-time* algorithms are algorithms that always try to improve the solution quality, for sensing processes, we can define deadline for the "searching" process every execution, ??? present any-time solution for image processing [31]. Set of *execution-modes* based, a discrete version influence from "contract-based" presented by ??? Pant [22], here each sensing task have finite number of "operation modes", e.g. different algorithms to proximate the same observations value, then the scheduler can perform the apropriate mode for every cycle.

For simplicity, we only work with *execution-modes* based sensing tasks, all the modes are pre-defined and identified by a pair $\langle Duration, Variance \rangle$, which define the error $Variance$ of this mode and the run-time $Duration$ is the of this mode. We assume that longer modes produce solution with smaller error variance.

4.2 State Estimator

State Estimator (the filter) is part of the estimation process, this unit receive the measurements (y) from the *Sensors* and produce from them the state estimation required by the controller ($\hat{x}$). When the *Sensors* is accurate enough we may be able to pass it directly to the *Controll Law* ($\hat{x} = y$), but usually the measurement is noisy and the main goal of the State Estimator is to reduce this noise from the measurement (see Section 12.1).

If one consider using the optimal estimator Kalman filter (describe at Section 12.1) in the estimation process, one of the parameter that need to be consider in calculation is the covariance of sensor error (noted by R in Section 12.1). But in the new framework the sensor have diferents operation modes with variable error covariance, this means that we need to have also variable state estimators correspondingly. In order to adjust the sensor error covariance, each cycle the scheduler will inform the state estimator about the new error covariance, and the state estimator will use corresponding parameters to make the next estimation.

4.3 Control Tasks

The control task (regulator) itself (*Control Law* in Figure 2) is responsible for closing the gap between current state estimation ($\hat{x}$) and the reference state (r), by manipulating the system actuators (e.g. changing the speed of the motors), that output to the actuators is noted by u.

The control task is usually a very low CPU consumer and been well studied [6, 7]. Hence, the control task is not needed to be manipulated. For our testing we use the

commonly used and well known technique from control theory called PID, A proportional-integral-derivative controller. In this technique the control output is based on the physical knowledge of the system dynamics, and have three variable parameters (P, I and D) that define the controller convergence behavior [**?**] ???.

4.4 Computation Resource Scheduler: Automata Based Scheduler

Real-time systems are mostly composed of multiple real-time tasks, tasks with time constraint, for example task that must response to an event within specified time constraints. The purpose of schedulers in such systems is to allocate the limited computational resources (CPU time) within all the task in the system. To do so, we need a well defined interface between the real-time tasks and the scheduler.

The most common way of describing the requirements of a real-time component is to specify a period, sometimes along with a deadline, which gives the frequency at which the component must execute. The designer of the component makes sure that the performance objectives are met as long as the component is executed consistent with its period. The scheduler guarantees that all components get enough resources. Specifying resource requirements using periods has advantages due to simplicity and analyzability, but has limited expressiveness, as elaborated in [2].

4.4.1 The Proposed Scheduling Methodology

In this thesis we develop a new methodology for allocation resources. We focus on how should engineers describe the requirements of a real-time component. To simplify, we will assume that the resource is allocated in discrete *time slots* of fixed duration in the style of time-triggered architecture [2]. The specification framework for resource requirements is based on *Nondeterministic ω-automata* (finite automata over infinite words) [**?**]. Automata can be more expressive for describing specific requirements, and they are composable, i.e., it is easy to compose all the tasks requirements into an integrated automata, and its easy to analyze and manipulate using tools like GOAL [**?**].

Formally *task specification automata* is defined, similarly to Nondeterministic ω-automata, as tuple $A = (Q, \Sigma, \Delta, Q_0, Acc)$. In our setting, the *alphabet* of the automata is $\Sigma = T^2 \times C^2$ where C is the set of Boolean conditions variables, will be describe later, and T is the set of all tasks in the system. Each infinite word ($\alpha \in \Sigma^\omega$) of the automata define a possible tasks scheduling over a fixed period (time slots), and the automata language define all the possible scheduling. Now the scheduler only need to "walk through" the automata as follows, before every iteration (time slot) the scheduler choose state transition $(q_i, \{t_j, c_j\}, q_{i+1})$ from the current state q_i to the state q_{i+1}

Let's define $s()$

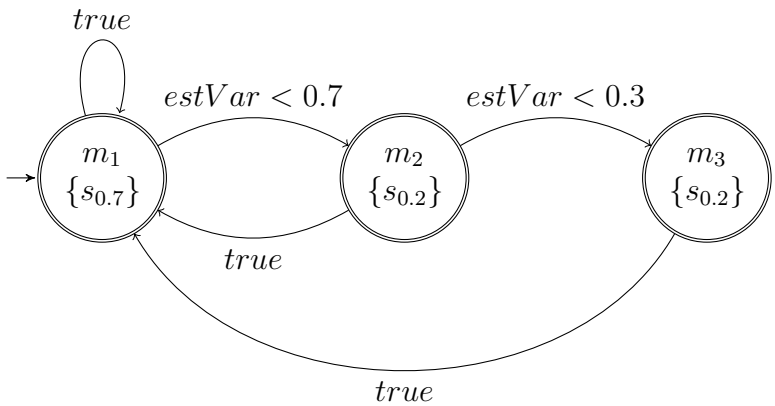


Figure 4: Example of guarded automata for our vision based sensor task, in this example the task has two operation modes, $\mathbf{s_{0.7}}$ which need 70% of the time slot but is more accurate vision computation and $\mathbf{s_{0.2}}$ which is less accurate but faster (need only 20% of the slot). The value of $estVar$ is $var(x - \tilde{x})$ of the previous iteration. Every time slot exactly one of the modes will be executed.

In our setting, the automata define continuous (fixed period) operation modes, and the condition of mode changing. The operation mode directly define a set of tasks that will be executed in the time slot (for example $s_{0.7}$ in Figure 4). We can stay at a single mode for some iterations, in this case the same tasks will be scheduled in each iteration. We can also take **discrete mode transition** in order to change operation mode after few iterations and schedule different set of tasks, for example, in Figure 4 the edge (m_1, m_2) says that if the previous iteration mode was m_1 and $estVar < 0.7$ we can change the mode and schedule the set $\{s_{0.2}\}$ in the next iteration.

Each infinite path on the composed hybrid automata (not necessarily with infinitely many mode transitions) represents a schedule that satisfy all the tasks requirements. In this architecture the scheduler only need to "walk through" the composed automata, this is, of course, a fast and easy computational task. In order to assure that we do not exceed the time slot duration, each task will have pre-defined maximum duration time like "deadline" in the traditional architecture. Now we can verify that every possible scheduling step can execute within a single time slot, in other words, every mode of the composed automata can be executed in a single time slot, by simply summing the "deadlines" of all the tasks in the mode tasks set. If we find a mode that goes beyond the maximum duration we can remove it from the automata so we never exceed the time slot duration.

Let us demonstrate the proposed automata based interface using the system depicted in Figure **??**. Assume we have two operation modes of the vision based sensor: (1) $s_{0.7}$ a very accurate operation mode that takes 70% of the time slot to execute, that is of course a significant amount of time, and (2) $s_{0.2}$ a less accurate operation mode but takes only 20% of the slot. In each time slot we can execute one of them and get the sensing process done, but if we use only $s_{0.7}$ every time slot we may not have enough time to execute all

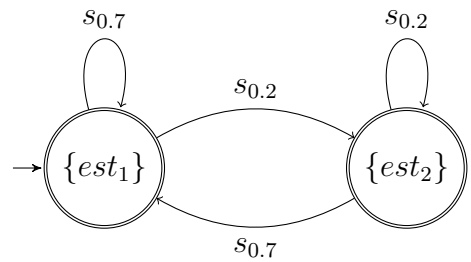


Figure 5: Example of guarded automata for our state estimator, in this example the estimator has two operation modes, $\mathbf{est_1}$ correspond to $\mathbf{s_{0.7}}$ and $\mathbf{est_2}$ correspond to $\mathbf{s_{0.2}}$ (see Figure 4).

the tasks. On the other hand, if we use only $s_{0.2}$ we will have inferior estimates. Figure 4 shows an example of a guarded automata that guides the schedule. With this automata we can express rich specifications. In this example, execute $s_{0.7}$ is always allowed but if we need faster sensing we can use $s_{0.2}$ but only once in a row if the estimation error is not extremely bad (if $var(x - \tilde{x}) < 0.7$), or even twice in a row if the estimation error is good (if $var(x - \tilde{x}) < 0.3$).

The estimated estimation error ($estVar$ in the figure) is, in this case, the estimation error variance ($var(x - \tilde{x})$ in Figure **??**).
This value ($estVar = var(x - \tilde{x})$) is calculated by the state estimator task (Section 4.2) and is passed to the scheduler as discussed before.

Each of this operation modes ($s_{0.7}$ and $s_{0.2}$) have different accuracy, specified by $var(v)$ in Figure **??**, and if we want to get optimal estimation the state estimator mast be configure correspondingly, i.e., the sensing error variance should be adjusted to the correct value ($var(s_{0.7})$), an easy solution for specifying the different configurations is by the guarded automata shown in Figure 5, which defines two operation modes of the state estimator, est_1 and est_2, that correspond to $var(s_{0.7})$ and $var(s_{0.2})$. So of course if we sense in mode $s_{0.7}$ we must estimate with mode est_1 that has the correct configurations for $s_{0.7}$, and if we sense in mode $s_{0.2}$ we must estimate with mode est_2.

5 A demonstration of the approach in simulation

Our approach for using automata for scheduling resources in software based controllers is based on the observation that in most systems the computational load is in the implementation of the sensors and actuators, not in the implementation of the controllers that usually consist of quick arithmetic manipulation of a small amount of variables. We therefore focus our attention on allowing a trade-off between CPU usage of sensors and actuators and their accuracy.

Formally, we assume that each of the physical processes we control is a Linear Time Invariant (LTI) system with a known model, as depicted in Figure 6. The inaccuracies

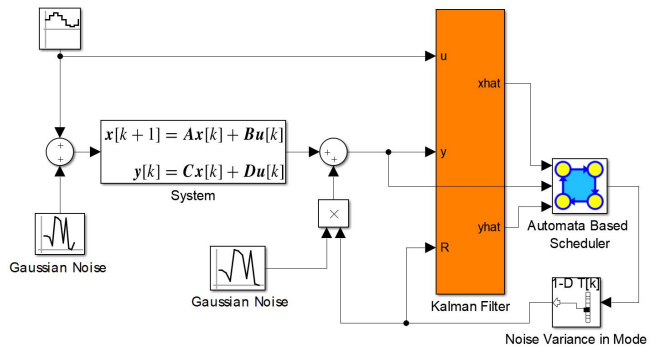


Figure 6: A Simulink model demonstrating the proposed approach.

of the sensors and of the actuators are modeled as additive Gaussian noise with a known variance. As a basis for scheduling, we assume that each sensor (and, possibly, also the actuators) can be scheduled to operate in one of a range of modes at each computation slot, each mode consuming a certain percentage of the CPU and giving a certain variance of the measurement (or actuation) noise.

The scheduling of the modes, for each of the components, is governed by the automata based scheduler as depicted in Figure 6. We propose to use a standard Kalman filter as a tool to gather the information that guides the state evolution of the automata, as follows. The filter gets as input the actuations, measurements, and the variance of the measurement noise. We assume that the measurament noise is a static function (represented here as a translation table) of the mode chosen by the scheduler. The output of the Kalman filter is fed to the scheduler that uses it for advancing the states of the automata. In the Simulink model depicted in Figure 6, the covariance matrix of the disturbance, which in this case is 1×1 matrix consisting of the noise variance, is fed to the Kalman filter (as its R input) and to the block that multiplies the Gaussian noise by the variance (the product of a white noise with unit variance and a constant v yields a Gaussian noise with variance v).

We ran the model depicted in Figure 6 with the linear time invariant system

$$\begin{aligned} x(k+1) &= \begin{pmatrix} 1.3 & -0.5 & 0.1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} x(k) + \begin{pmatrix} -0.4 \\ 0.6 \\ 0.5 \end{pmatrix} u(k) \\ y(k) &= \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} x(k) \end{aligned}$$

taken from `mathworks.com/help/control/examples/kalman-filter-design.html`. As seen in Figure 6, we injected a sinusoidal input (with $amplitude = bias = frequency = 1$) to this system. The actuation noise, depicted on the left, is with a unit variance.

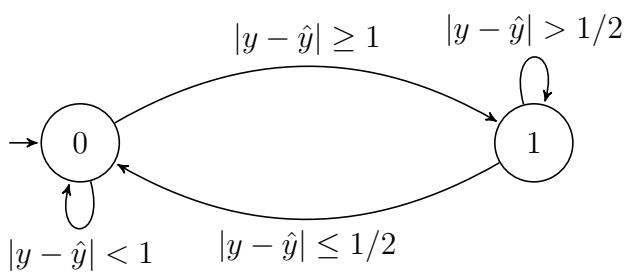
The 'Automata Based Scheduler' block is designed to set the variance of the sensing noise dynamically to be either 0.25 or 1 at each step of the simulation. This models a sensor that has two modes of operation: a mode with high accuracy, that produces normally distributed measurement errors with low variance, and a mode with low accuracy

	High	Low	Aut. Based
%CPU	85	10	46
mean of $\lvert x - \hat{x} \rvert$	0.97	1.24	1.08

Table 1: Simulation results.

that produces normally distributed measurement errors with low variance. We assume, for the performance measurements presented below, that the CPU consumption of each mode is $\%CPU = 1.1 - errVar$, where $errVar$ is the variance of measurement errors in the mode.

We ran this model with three versions of the 'Automata Based Scheduler' block. The first version, called 'High' in Table 1, is where the block acts simply as the constant 1, ignoring its inputs altogether. Similarly, the term 'Low' in the table refers to an implementation where the block is the constant 0. These two implementations model the constant schedules, where the sensor is operated in one mode along the whole execution. These two schedules are compared to a third implementation, called 'Aut. Based' in the table, where the block implements the schedule given by the automaton:



The results of the simulation, summarized in Table 1, show, as expected, that the CPU consumption is much lower (0.1) when using the low-quality version of the sensing algorithm and is higher (0.85) when the high-quality version of the sensing algorithm is used. The performance of the estimation, in terms of the mean distance between x and $\hat{x}$, is better with the high-quality version (0.97) than it is with the low-quality version (1.24). More interestingly, we can see that the experiment with the automaton that switches between the two sensor modes yields performance that is close to the performance of the high-quality sensing algorithm, using much less CPU. This demonstrates how automata based reactive scheduling can allow for new ways to balance performance and computational resources in software based controllers.

6 Case Study: Stabilizing a quadrotor in front of a window

The case study we used to test our concept is the development of a controller that stabilizes a quadrotor in front of a window (see, e.g., `rpg.ifi.uzh.ch/aggressive_flight.html`).

We implemented an autonomous controller for that task and evaluated its performance.

The part of the controller that we focused on is the vision based position controller. Specifically, the main controller, that we will describe below, uses a standard low-level angular controller and a simple image processing algorithm that identifies the position of the corners of the window in the image plane[1]. Its goal is to regulate the position of the quadrotor by tilting it. Note that rotations of the quadrotor generate a more-or-less proportional accelerations in a corresponding direction, hence we can approximate the system with a linear model. A main challenge for this controller is that the computer vision algorithm takes significant time to compute relative to the fast control loop. We can decrease computation time by lowering the resolution, but this also increases the measurement noise. We will demonstrate how reactive scheduling of the resolution can serve for balancing resource consumption vs. control performance in this system.

6.1 Observer Design

We first implemented an observer based on the work of Efraim [12]. The observer gets the positions of the widow corners from the image (see Section 9.1 for more details), and extracts the following four quantities based on the shape and location of the window corners in the image plane: **Center of mass:** S_x and S_y represent the window "center of mass" along the x and y axes of the image, respectively, normalized to the range of $[-1, 1]$. These values are used for controlling the roll angle and the altitude of the drone, respectively. **Window size:** sz is the sum of the vertical edges of the window. It is used to measure the distance of the drone from the window [2]. **Vertical difference:** $V_d = ((y_1 - y_4) - (y_2 - y_3))/((y_1 - y_4) + (y_2 - y_3))$, were y_i is the vertical position of the widow corners in the image in the range of $[-1, 1]$, enumerated clockwise starting from the top left corner. It is used to measure the angular position of the drone in relation to the window.

As common in engineering practices, we pass these numbers to a filter that gives us a more informed state estimation. In theory, as demonstrated in the simulation in Section 5, we should use Kalman filter estimator for best state estimation. As we deal with a non-linear system here, and because the process noise distribution is not known (it is significantly affected by the varying state of the battery), and because Kalman filter adds complexity in the code, we propose to use a complementary filter. A complementary filter is actually a steady-state Kalman filter for a certain class of filtering problems [16]. Specifically, in our case study, we implemented a two steps estimator that: (1) *predicts* the current state evolved from the previous state (denoted by $\hat{x}_{k|k-1}$) using a linearized model of the system, and then *updates* the prediction with current state measurement from the

[1] In the experiment, to simplify the image processing algorithm, we marked the corners of the window with led lights.

[2] We used a fixed size window and converted sz to distance (in meters in our case) based on the size of the window. In the general case the units of distance are relative to the window size.

image, denoted by y_k. The result of the estimation, denoted by $\hat{x}_{k|k}$, is a complementary filter of the prediction ($\hat{x}_{k|k-1}$) and the measured state ($C^{-1}y_k$) [3]: $\hat{x}_{k|k} = K\hat{x}_{k|k-1} + (1 - K)C^{-1}y_k$ were K is constant that equals the Kalman filter gain (K_k) at a steady-state.

The image processing algorithms can run in different operation modes, each mode with different accuracy (see Section 6.3). The constant K represents the ratio between the variance of the process noise and the variance of the measurement noise. To achieve best performance, we define a separate value of K for each mode.

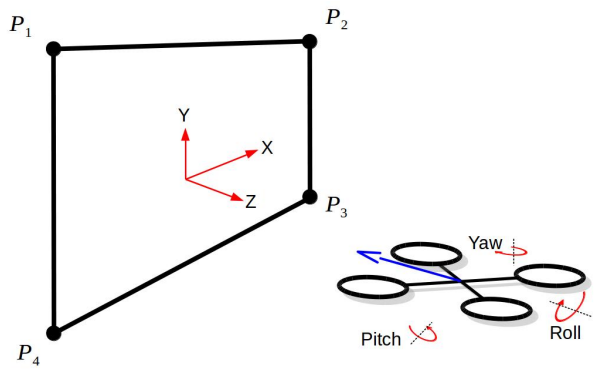


Figure 7: The coordinate system and the rotation axes of the drone.

6.2 Controller Design

In this experiment, we consider the task of hovering in front of the window. The controller objective is to hover parallel to the window (center of z axis in Figure 7) at the altitude of the window within distance of two meters in front of the window ($y = x = 0, z = 2$) and face pointed to the center of the window (yaw angle). We consider this point as the origin and the coordinates are according to Figure 7.

The controller consists of four independent feedback loops: altitude, yaw, pitch, and roll. The pitch and the roll controllers are copies of the same attitude controller which gets as input the required pitch or roll angel, respectively. These controllers runs in parallel to a position controller that maintains the required distance (z position) and displacement (x position) relative to the window, the position level controller outputs the required angle (acceleration) to the attitude controller as show in Figure 8. All the control loops regulate the position or attitude relative to the window. The inertial (pitch and roll) feedback for the low level controller is generated by the existing *Attitude and Heading Reference system* (AHRS) library of ArduPilot (see Section 9.1) and the position feedback (x, y, and z position) came directly from the observer described in Section 6.1.

Based on the *principle of separation of estimation and control*, we can use that controller regardless of the measurement quality. We implemented a basic Proportional Integral Derivative (PID [4]) controller and tuned the parameters by trial and error using the

[3]The matrix C is the measurement matrix as shown in the model at Figure 6

highest resolution observer.

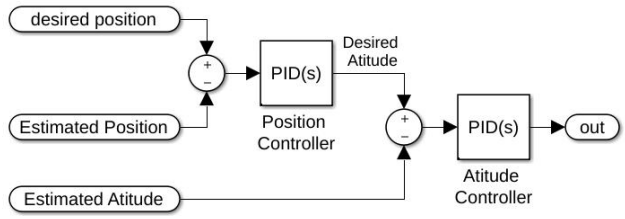


Figure 8: Attitude and position controller consisting of a two level, cascaded, structure.

6.3 Analysis and Specification Automata

Since the objective of the system is to maintain stable hovering in front of the window, performances is measured by the amount of deviation from the center line of the window in the critical axis x, i.e., we want to minimize $|x|$ (see Figure 7). Our goal is to achieve maximum performance with minimal amount of processing time. Obviously, both goals cannot be achieved together. We therefor aim at achieving a good trade-off between processing resources and quality of measurement. The focus is on the main consumer of computation resources: the image processing algorithm. We aim at taking resources (high CPU time) only when needed. By this, we will reduce the average CPU usage without significantly affecting the performance. Specifically in our case study, we switch between camera resolutions dynamically during the flight. The observer modes, discussed above, correspond to image resolutions (see Section 6.1). For simplicity, we consider in this paper only two modes: *High quality* with resolution of 960p and *Low quality* with resolution of 240p.

Looking at the test results shown in Table 2, we see that the high quality observation mode provides mean error tolerance of $9.5cm$ [4] (in the x axis) with the cost of 30% CPU usage. On the other hand, the low quality mode provides mean error of $30cm$ (three times the error obtained in the high quality mode) using only 2.1% CPU usage [5].

We explain next several approaches we applied in the design of a *reactive scheduling specification* that combines the advantages of both the high and the low resolution sensing modes.

We first examined a straight-forward scheduler based directly on the current position (the x axis of $\hat{x}_{k|k}$). Using the simple guarded automaton A_x presented in Figure 10, we defined a dynamic observer that chooses the mode (high or low sensing quality) based on the absolute value of the x axis of $\hat{x}_{k|k}$. If the drone is closer to the center line of the window, denoted by $|x| < T_x$, we consider it as a "safe" state and go to the mode called

[4] All lengths are measured in centimeter.

[5] The CPU usage percentage is the average: (total time spend in image processing)/(total flight time).

L that schedules the low quality image. Similarly, a state is considered "BAD" when the drone is far, denoted by $|x| > T_x$. In this case, we take high quality images in order to "fix" the state. The threshold T_x can be changed to tune the desired performances and processing time trade-off. See Section 7 for the data we collected by experiments to validate that this indeed achieves the required goals.

In a second experiments set, we used the A_{err} automaton presented in Figure 10. This automaton is similar to A_x with the difference that it switches states based on $\tilde{y}_{k|k}$ value defined as:

Definition 1 *Measurement post-fit residual $\tilde{y}_{k|k}$, is the difference between the expected measurement, $C\hat{x}_{k|k}$, and the measured state, y_k,: $\tilde{y}_{k|k} = |C\hat{x}_{k|k} - y_k|$.*

The motivation to use this indicator comes form looking at the low quality experiment graph, shown in Figure 9. We can see that $\tilde{y}_{k|k}$ growth is proportional to the deviation from the center of the window (in the x axis). We can also see that the growth in $\tilde{y}_{k|k}$ precedes the growth in x. This means that we can use $\tilde{y}_{k|k}$ values to predict deviations in x.

In A_{err} the observer operates in low quality when $\tilde{y}_{k|k} < T_l$ ("GOOD" state) and switches to high quality when $\tilde{y}_{k|k} > T_h$ ("BAD" state). The thresholds T_l and T_h were initially taken from the low and the high quality graphs and fine tuned by experiments. Different T_l and T_h give different trade-offs of performances and processing time. In Section 7 we show that this dynamic observer gives almost the same performance as the high quality observer with a reduction of a factor of two in the processing load.

The expressiveness of automata allows for creating even more complex scheduling specification. We can, for example, combine the approaches of both A_{err} and A_x. In Figure 10 there is two examples of such combinations. A_{comp1} tries to save unnecessary CPU usage while the drone is in "safe" state, close to the center. It activates A_{err} only if the drone goes far from the center ($T_x$cm from the origin). A_{comp2} adds another constraint: if the drone goes even farther away (T_{x2}cm from the origin) it schedules only high resolution to bring it back to safety. The results show even better resource utilization.

7 Results

Table 2 summarizes the result of the flight experiments made using the specification automata described in Section 6.3. The first column "% CPU" represents processing time usage by the image processing task. The second column "*mean*(|x|)" is the average distance of the drone from the origin in centimeters. All the statistics shown in the table are from real indoor experiment flights. Each experiment consists of a flight (between one and two minutes) recorded using an *OptiTrack* system (see Section 9.1) and data from the software itself.

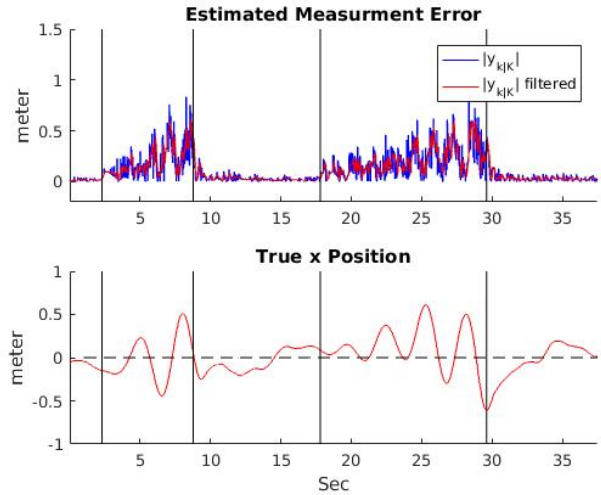


Figure 9: This figure shows in the bottom plot the position of the quadrotor during the flight time and in the top plot the $\left|\tilde{y}_{k|k}\right|$ value during the flight. During the flight we manually switched between low and high resolution for further analysis. The modes (high or low) are separated by vertical lines, were the leftmost is high resolution then low and so on.

We see that, using reactive scheduling methods, we can adjust the trade-off between performance and CPU usage. As expected, if we use **only high** resolution mode we get the best performance but need much %CPU. On the other hand, the **only low** resolution mode barely use the processor but the performance are low and unsatisfying for indoor flights.

In Table 2, automata number 3,4 and 5 are the implementation of A_x automaton as describe in Section 6.3 with the threshold values of 10, 20 and 30cm respectively. Those schedulers demonstrate that even a simple scheduling scheme, that schedules the expensive image processing task only when it is most needed, improves the performance. Scheduler number 3 achieves almost the same performance as the only high resolution scheduler but saves half of the CPU time. Scheduler automata 4 and 5 provide different trade-offs.

The A_{err} automaton (shown in Figure 10), which bases its transitions on $\tilde{y}_{k|k}$, appears to be even more effective. We tested this automaton with $(T_l = 10, T_h = 20)$ and $(T_l = 10, T_h = 15)$ thresholds as shown by schedulers 6 and 7 respectively in Table 2. They are using even less CPU time but still achieve fairly impressive performance. The more complex schedulers contribute even more to the performance and utilization. Scheduler A_{comp1} for example achieves a little better resource utilization. The methodology we are proposing is to tune the thresholds and to enrich the automata, as we did, until satisfactory CPU utilization and performance are achieved.

Another advantage we achieved using the dynamic schedulers, beyond allowing better performance to computation load balance, is reactiveness. The scheduling scheme we are proposing allows the system to adapts itself to changes in environmental conditions, demonstrated as follows. We experimented in exposing the flying drone to time-varying

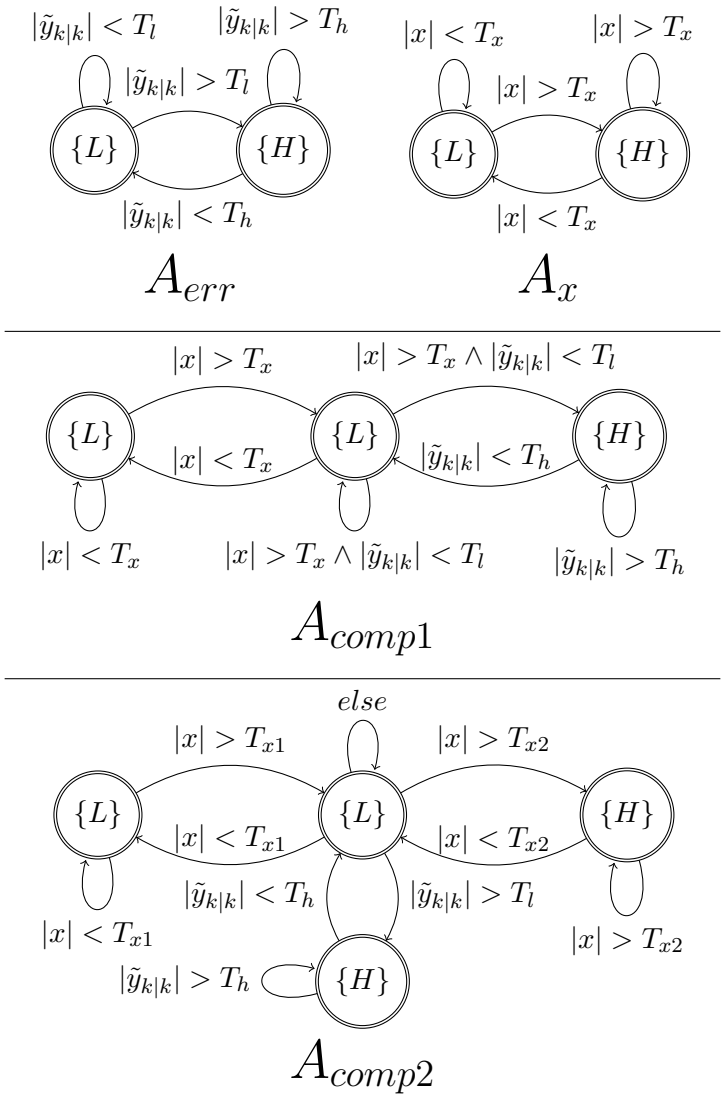


Figure 10: The main Automata we used for the requirements specifications in the experiments.
A_{err} uses the *measurement post-fit residual* ($\tilde{y}_{k|k}$), if $|\tilde{y}_{k|k}|$ is too small then the next iteration will use small resolution image (L), and if it is too large the high resolution will be used, T_l and T_h are the transition thresholds. A_x operates similar to A_{err}, only it uses the current position (the x part of $\hat{x}_{k|k}$) instead of $\tilde{y}_{k|k}$, with the threshold T_x. A_{comp1} and A_{comp2} area composition of the ideas in A_{err} and in A_x.

wind conditions. Specifically, we created artificial wind by turning on a fan while the drone was flying. The scheduler was the A_{err} automaton with the thresholds $T_l = 10$ and $T_h = 15$ (number 7 in Table 2). Figure 11 shows data collected in one of the flights with the CPU usage and the displacement of the drone along the flight. The first part of the flight, colored in blue, is indoor flight without any external wind. The second part, colored in red, is a flight with the disturbance caused by the wind of the fan. We can clearly see increasing CPU usage when there is wind. We can also see that the wind does not affect the displacement, as the increased accuracy of the sensor compensates for it. The mean displacement ($mean(|x|)$ in this experiment was 11.8cm which is very close to the displacement without the wind, and the average CPU usage increased to 13.2% from 11.7%. This experiment further demonstrates our idea: resources can be allocated only when needed.

Table 2: Test Results

	Schedule	**% CPU**	**mean(—x—) (cm)**
(1)	Only High	30.9%	9.5
(2)	Only Low	2.1%	30.0
(3)	A_x ($T_x = 10$)	16.6%	10.9
(4)	A_x ($T_x = 20$)	14.0%	14.1
(5)	A_x ($T_x = 30$)	8.9%	17.4
(6)	A_{err} ($T_l = 10, T_h = 20$)	10.3%	14.9
(7)	A_{err} ($T_l = 10, T_h = 15$)	11.7%	11.3
(8)	A_{comp1} ($T_x = 10$, $T_l = 10$, $T_h = 15$)	8.8%	12.9
(9)	A_{comp2} ($T_{x1} = 10$, $T_{x2} = 30$, $T_l = 10$, $T_h = 15$)	10.4%	12.7

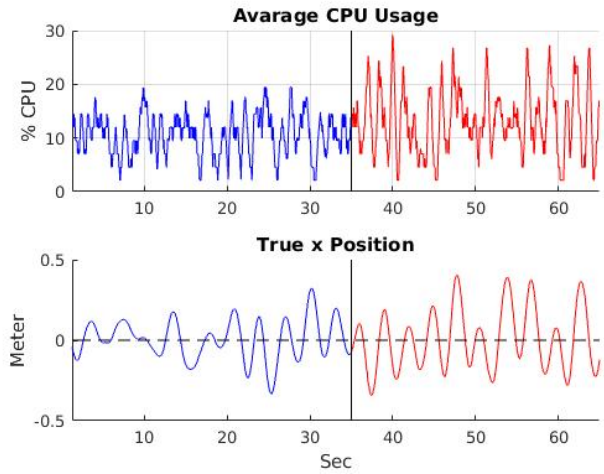


Figure 11: This figure demonstrates how the reactive scheduler adapt to the environment changes, in particular, to change of the wind. The upper part shows the average CPU usage during the flight, and the bottom is the true position at the same time. The first part of the flight (the left Blue graph) is without any external wind and in the rest of the flight (right red graph) the drone was exposed to artificial wind (using a fan). Note that when the drone is exposed to wind it uses more CPU to compensate for the disturbance.

8 Scheduler Prototype

8.1 Formal Definition of Guarded (specification) automata

8.2 Automata Operation

8.3 GOAL Tool

9 Testing Environment

9.1 Experiment setup (From Paper)

In order to perform the experiments we used an of-the-shelf quadrotor with a *raspberry-pi* (www.raspberrypi.org) board with the *navio2* (emlid.com/navio/) shield. We implemented the controllers and the schedulers as libraries for the ArduPilot Mega (APM) autopilot software.

We used the standard raspberry-pi camera with a modified driver to allow for dynamic resolution switching. In order to simplify the image processing algorithm, we marked the four corners of the window with four illuminated balls. This allowed us to apply a simple minded corner detection algorithm: look for a blob of burned pixels and calculate the "center of mass" of all those pixels as a window corner. Higher resolution give us a more accurate measurement of corner "center of mass".

In order to evaluate the performance, we needed to know the accurate position of the

drone during the flight. For this purpose, we used the *OptiTrack* (optitrack.com) system consisting of an array of high-speed tracking cameras that provide low latency and high accuracy position and attitude tracking of the drone. The data from the *OptiTrack* system is considered in this paper as the ground truth.

9.2 APM Controlled Quad-copter

9.3 Vision based Autonomous In-Door Flying

Computer Vision Based Sensor is the module that is responsible for taking a picture or a series of pictures and produce measurements of quantities such as speed and position relative to the environment. In our research we will use the *Computer Vision* in order to detect two dimensional movement in the camera surface (i.e., the drone speed). There are many such algorithms (called optical flow) differing in running times and in accuracy. Usually more invested time leads to more accuracy.

In our case we need to be able to control the *Computer Vision* running time and to have some good knowledge of the solution accuracy in order to achieve optimal state estimation (see Section 4.2). We assume that the *Computer Vision* error is distributed normally, and we will use the error variance as a measure of accuracy. We believe that this is a reasonable assumption because the estimated speed is usually computed as the avarage of of many independent random variables, as follows. Optic flow algorithms usually go by identifying similar regions in consecutive pictures and then averaging the distances that each feature "traveled". Assuming that the error in measurement of each feature is independent of the other errors, we get that the total error is the average of independent random variables. Then, by the law of large numbers, we get that the error should have a normal distribution. We will validate this assumption by experimentation with different parameters of different algorithms.

In order to control the running time, we will use anytime based algorithms [31] and will mainly concentrate on "contract based" vision algorithms proposed by Pant [22]. This type of algorithms run until we stop them, and when we stop them they will provide a solution with accuracy that is a monotonic function of the amount of time it was running. That way we can control the solution accuracy by controlling the running time. In our implementation, in order to lower the complexity, we will pre-define few specific "operation modes" of the *Computer Vision* task, that differ by their running time and they are identified by a pair $\langle RunTime, Variance \rangle$ where $Variance$ is the error variance of running time $RunTime$.

9.4 Reference Measurement with OptiTrack System

10 Related Work

11 Conclusions and Future Work

We introduced a new approach for resource allocation that allows dynamic scheduling where resources are only allocated when needed.

We presented a compositional design strategy using rich automata based interface between software components and the task scheduler. We also proposed how to use this design in the context of software control systems using the data provided by standard filters. We demonstrated the advantages of the resulting dynamic schedulers in terms of control performance and resource utilization. These advantages are demonstrated both in simulations and with a real case-study.

The use of automata allows to compose and to analyze the scheduling characteristics. As future work, we plan to apply automata theory and, more specifically, the theory of hybrid automata to develop new techniques and tools for automatic generation of specification automata. The input for this construction can be a specification of the dynamical system that we want to control and a specification of the required close-loop characteristics.

12 Apendix

12.1 Kalman Filter and State Estimation

Te role of state estimation as the name suggest is to estimate the current state of the system, usually represented as the vector x. One can monitor the system with an array of sensors, the measurement devices intend to be uncertain, the measurement vector noted by $y = Cx + v$ where v represent the measurement error. Another estimation technique would be prediction, the system dynamics are known and the system inputs (u) are known, then create a model of the system using physics equations to predict the system state evolution in time (assume the initial state is known). The predicted state, noted by $\hat{x}$, is also not precise the real world is too complex to express in reasonable way, and we mark the predicted error by w.

In order to improve the estimation certainty a *filter* is added to the state estimation process, the filter aggregate the sensors and produce better estimation of the state. We will cover two well known filters, *complementary filter* and *Kalman filter*.

A short brief on *complementary filter*, this is basically combination of two filters (witch are complement each other), in control field it usually refers to *Low-Pass filter* and *High-Pass filter*. Low-pass filter allows the low frequencies signals to pass and filter out the high

frequencies signals, is used for signals with high frequency error, e.g accelerometer signal. high-pass filter is its complementary, meaning it filters out the low frequencies signals, is used for signals like gyroscope that have continuous accumulated error.
General complementary filter notation: $\hat{f} = \alpha \cdot f_h + (1 - \alpha) \cdot f_l$ s.t. $\alpha \in [0, 1]$
if α is significant high, f_l is considered *low-pass*, as rapid value changes less significant and d slow value changes can be accumulated over time, and f_h is considered *high-pass*.
In vision base measurements we use the only Low-pass filter: $\hat{x}_k = \hat{x}_{k-1} + \alpha \cdot (y_k - \hat{x}_{k-1})$.

Complementary and Low-pass are simple and very easy to implement but they not necessary produce the best possible estimation. A well known estimator called *Kalman filter* is a more complex but optimal filter, Kalman filter assume linear system ($x_k = Ax_{k-1} + Bu_k + w_k$) and zero mean Gaussian errors with covariance matrices $cov(w_k) = Q$ and $cov(v_k) = R$, if those conditions holds kalman filter produce the best possible estimation, based on all previous information available.

This is recursive algorithm that works in a two-step process, *prediction* step and *update* step. In the prediction step, the system model used to predict the current state, this called *A Priory Estimate* and noted by $\hat{x}_k^-$, the error covariance P_k^- of the prediction is also calculated. Then in the *update* step the prediction is updated to find out the optimal estimation called *A Posteriori Estimate* noted by $\hat{x}_k$) and it's error covariance P_k. Practically the update step is a weighted average between the prediction (P_k^-) and the measurement (y_k), with more weight being given to estimates with higher certainty [17].

assumes the true state at time k is evolved from the state at ($k1$) according to:

$$x_k = Ax_{k-1} + Bu_k + w_k$$

and the measurement at time K is:

$$y_k = Cx_k + v_k$$

Where x_k is the system state at time step k and y_k is the measurement by the sensors at time step k. w_k and v_k represent the process and measurement noise with covariance matrices Q and R. The system state at time $k - 1$ (x_{k-1}) is unknown, therefore the previous estimation is used to predict the next step state using the recursive equation:

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_k$$

the error covariance of the prediction composed of the error of previous estimation which is: $cov(A\hat{x}_{k-1} - x_{k-1}) = Acov(\hat{x}_{k-1} - x_{k-1})A^T = AP_{k-1}A^T$ and the process error of time k (Q) and we get prediction error covariance of:

$$P_k^- = AP_{k-1}A^T + Q$$

After the prediction was calculated the update step combine the prediction and measurements to get the Kalman estimation:

$$\hat{x}_k = \hat{x}_k^- + K_k(y_k - C\hat{x}_k^-)$$

K_k, also called Kalman Gain, represent contribution each partial estimation ($\hat{x}_k^-$ and y_k) to the Kalman estimation, and is calculate each time by:

$$K_k = \frac{P_k^- C^T}{CP_k^- C^T + R}$$

And the Kalman estimation covariance is:

$$P_k = (I - K_k C)P_k^-$$

The Kalman filter is designed for linear systems. It is not suitable in the case of a nonlinear systems, if the state transition function or the measurement function are nonlinear:

$$x_k = f(x_{k-1}, u_k) + w_k$$

$$y_k = h(x_k) + v_k$$

A Kalman filter that linearizes the nonlinear function around the mean of current state estimation is used, this referred to as an extended Kalman filter (EKF). The transition and mesurment functions must be differentiable in order to linearize them, and the Jacobian matrices are used. Extended Kalman filter is not guaranty to be optimal filter.

12.2 Raspberry-Pi Camera Driver

12.3 TODO

12.4 TODO

References

[1] R. Alur and G. Weiss. Regular specifications of resource requirements for embedded control software. In *Proceedings of the 14th IEEE Real-time and Embedded Technology and Applications*, 2008.

[2] Rajeev Alur and Gera Weiss. Rtcomposer: A framework for real-time components with scheduling interfaces. *8th ACM/IEEE Conference on Embedded Software*, 2008.

[3] K-E Arzén, Anton Cervin, Johan Eker, and Lui Sha. An introduction to control and scheduling co-design. In *Decision and Control, 2000. Proceedings of the 39th IEEE Conference on*, volume 5, pages 4865–4870. IEEE, 2000.

[4] Karl Johan Åström and Tore Hägglund. *Advanced PID control*. ISA-The Instrumentation, Systems and Automation Society, 2006.

[5] Joshua Auerbach, David F. Bacon, Daniel T. Iercan, Christoph M. Kirsch, V. T. Rajan, Harald Roeck, and Rainer Trummer. Java takes flight: time-portable real-time programming with exotasks. In *Proceedings of the conference on Languages, Compilers, and Tools for Embedded Systems*, pages 51–62, 2007.

[6] Stuart Bennett. A brief history of automatic control. *IEEE Control Systems Magazine*, 16:17–25, 1996.

[7] Anton Cervin, Dan Henriksson, Bo Lincoln, Johan Eker, and Karl-Erik rzn. How does control timing affect performance?, 2003.

[8] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and M. Stoelinga. Resource interfaces. In *Embedded Software, 3rd International Conference*, LNCS 2855, pages 117–133, 2003.

[9] Aveek K Das, Rafael Fierro, Vijay Kumar, James P Ostrowski, John Spletzer, and Camillo J Taylor. A vision-based formation control framework. *IEEE transactions on robotics and automation*, 18(5):813–825, 2002.

[10] L. de Alfaro and T.A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.

[11] Ardupilot Developers. ArduPilot Mega (APM): open source controller for quadcopters. www.arducopter.co.uk.

[12] Hanoch Efraim. *Vision Based Control of Micro Aerial Vehicles in Indoor Environments*. PhD thesis, Ben Gurion University of The Negev, 2017.

[13] Hanoch Efraim, Shai Arogeti, Amir Shapiro, and Gera Weiss. Vision based output feedback control of micro aerial vehicles in indoor environments. *Journal of Intelligent & Robotic Systems*, 87(1):169–186, Jul 2017.

[14] M. Esnaashari and M.R. Meybodi. A learning automata based scheduling solution to the dynamic point coverage problem in wireless sensor networks. *Computer Networks*, 54(14):2410 – 2438, 2010.

[15] T.A. Henzinger and J. Sifakis. The embedded systems design challenge. In *FM 2006: 14th International Symposium on Formal Methods*, LNCS 4085, pages 1–15, 2006.

[16] Walter T Higgins. A comparison of complementary and kalman filtering. *IEEE Transactions on Aerospace and Electronic Systems*, (3):321–325, 1975.

[17] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[18] E.A. Lee. What's ahead for embedded software. *IEEE Computer*, pages 18–26, September 2000.

[19] Edward A Lee. Cyber physical systems: Design challenges. In *Object oriented real-time distributed computing (isorc), 2008 11th ieee international symposium on*, pages 363–369. IEEE, 2008.

[20] Jun Liu, Necmiye Ozay, Ufuk Topcu, and Richard M Murray. Synthesis of reactive switching protocols from temporal logic specifications. *IEEE Transactions on Automatic Control*, 58(7):1771–1785, 2013.

[21] A.K. Mok and A.X. Feng. Towards compositionality in real-time resource partitioning based on regularity bounds. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 129–138, 2001.

[22] Yash Vardhan Pant, Kartik Mohta, Houssam Abbas, Truong X. Nghiem, Joseph Devietti, and Rahul Mangharam. Co-design of anytime computation and robust control (supplemental). Technical Report UPenn-ESE-15-324, Department of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA, May 2015.

[23] J. Regehr and J.A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 3–14, 2001.

[24] A.L. Sangiovanni-Vincetelli, L.P. Carloni, F. De Bernardinis, and M. Sgori. Benefits and challenges for platform-based design. In *Proceedings of the 41th ACM Design Automation Conference*, pages 409–414, 2004.

[25] S. Sastry, J. Sztipanovits, R. Bajcsy, and H. Gill. Modeling and design of embedded software. *Proceedings of the IEEE*, 91(1), 2003.

[26] Omid Shakernia, Yi Ma, T John Koo, and Shankar Sastry. Landing an unmanned air vehicle: Vision based motion estimation and nonlinear control. *Asian journal of control*, 1(3):128–145, 1999.

[27] Insik Shin and Insup Lee. Compositional real-time scheduling framework with periodic model. *Trans. on Embedded Computing Sys.*, 7(3):1–39, 2008.

[28] Paulo Tabuada. Event-triggered real-time scheduling of stabilizing control tasks. *IEEE Transactions on Automatic Control*, 52(9):1680–1685, 2007.

[29] L. Thiele, E. Wanderer, and N. Stoimenov. Real-time interfaces for composing real-time systems. In *Proceedings of the 6th ACM/IEEE International Conference on Embedded Software*, pages 34–43, 2006.

[30] Gera Weiss and Rajeev Alur. Automata based interfaces for control and scheduling. In *International Workshop on Hybrid Systems: Computation and Control*, pages 601–613. Springer, 2007.

[31] Shlomo Zilberstein and Stuart J. Russell. Optimal composition of real-time systems. *Artificial Intelligence*, 82(1-2):181–213, 1996.