

BEN-GURION UNIVERSITY OF THE NEGEV  
THE FACULTY OF NATURAL SCIENCES  
DEPARTMENT OF COMPUTER SCIENCE

Computational Resource Management of Multi-Channel Controller

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE MASTER OF SCIENCES DEGREE

Hodai Goldman

UNDER THE SUPERVISION OF: Dr. Gera Weiss

November 2017



BEN - GURION UNIVERSITY OF THE NEGEV

THE FACULTY OF NATURAL SCIENCES  
DEPARTMENT OF COMPUTER SCIENCE

COMPUTATIONAL RESOURCE MANAGEMENT OF MULTI-CHANNEL  
CONTROLLER

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE MASTER OF SCIENCES DEGREE

HODAI GOLDMAN  
UNDER THE SUPERVISION OF: DR. GERA WEISS

Signature of student: \_\_\_\_\_

Date: \_\_\_\_\_

Signature of supervisor: \_\_\_\_\_

Date: \_\_\_\_\_

Signature of chairperson  
of the committee for graduate studies: \_\_\_\_\_

Date: \_\_\_\_\_

October, 2014



# Abstract

We present an approach to reactive scheduling of computations in software control systems. The main motivation comes from the growing use of computer vision algorithms in real-time, as sensors. The term reactive here refers to the ability of the scheduler to adapt the schedules dynamically based on physical conditions. We propose an extension of the automata based scheduling approach with an addition of guards to transitions that allow for reactive specifications. We develop a methodology for using Kalman filters to provide data that guides these automata, and demonstrate the combined approach in simulations and with a case study. The case study is a development of a software that stabilizes a drone in front of a window using a vision based sensor with a time-varying resolution, i.e., computation load is controlled by taking images in reduced resolution when the state of the controlled loop allows.

abstract from  
paper

need to update Acknowledgments section

## Acknowledgments

The acknowledgments.

# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
<b>2</b>	<b>Problem Statement</b>	<b>10</b>
<b>3</b>	<b>The proposed approach</b>	<b>11</b>
3.1	Architecture . . . . .	12
3.2	Guarded Automata as Interfaces for Control and Scheduling . . . . .	12
3.3	Implementation in an Auto Pilot Software . . . . .	13
3.4	Integration With a Kalman Filter . . . . .	14
<b>4</b>	<b>A demonstration of the approach in simulation</b>	<b>14</b>
<b>5</b>	<b>Case Study: Stabilizing a Quad-rotor in Front of a Findow</b>	<b>17</b>
5.1	Observer Design . . . . .	18
5.2	Controller Design . . . . .	19
5.3	Analysis and Specification Automata . . . . .	21
<b>6</b>	<b>Results</b>	<b>23</b>
<b>7</b>	<b>Experiment Setup</b>	<b>25</b>
7.1	Hardware - the Quad-Rotor . . . . .	25
7.2	Vision Based Sensor . . . . .	27
7.3	Reference Measurement with OptiTrack System . . . . .	28
<b>8</b>	<b>Scheduler Interface Prototype</b>	<b>29</b>
8.1	A Schedule Component . . . . .	29
8.2	Component Games Composition . . . . .	31
8.3	Add Resources Limitation . . . . .	31
8.4	Wining the Game - Find the Strategy . . . . .	32
8.5	Execution of the scheduler . . . . .	32
8.6	Correctness . . . . .	32
<b>9</b>	<b>Few Words about Implementation</b>	<b>35</b>
<b>10</b>	<b>Related Work</b>	<b>35</b>
<b>11</b>	<b>Conclusions and Future Work</b>	<b>35</b>
<b>12</b>	<b>Apéndix</b>	<b>36</b>
12.1	Kalman Filter and State Estimation . . . . .	36
12.2	Raspberry-Pi Camera Driver . . . . .	38

12.3 Complementary-Kalman design (with physical computation) . . . . .	38
12.4 TODO . . . . .	38

# List of Figures

1	A typical feedback control loop where the physical plant ( <i>System</i> ) is monitored via an array of sensors ( <i>Sensing</i> ) which produce noisy sample of the state variables $y$ . And after sensing, <i>State Estimator</i> aggregates the measurements from all the sensors and produce estimation of the current state $\hat{x}$ . Then the <i>Control Law</i> produce output to the actuators ( $u$ ) in order to close the gap between the current estimated state and the reference state ( $r$ ). . . . . .	11
2	The controller framework we implement, Each control loop (depicted in blue) informs the resource allocator (Scheduler) of its quality of estimation, the <i>scheduler</i> will allocate CPU time accordingly in order to maintain valid estimation quality. Specifically for the Sensing processes, for example in our case <i>Computer Vision</i> task, the guard automaton specify the required sensing quality ( $var(v)$ ) which determines the required execution time ( $time$ ). The Sensing alocation is noted by $\langle time, var(v) \rangle$ . . . . .	13
3	APM scheduling specification . . . . .	14
4	A Simulink model used to demonstrating and examine the proposed approach. . . . .	15
5	Geometric relation between $\alpha$ (the angle derived from $V_d$ ) $l$ (distance from window extracted from $sz$ ) and the position of the drone among the $x$ axis. (TODO - need to replace image) . . . . .	18
6	The coordinate system and the rotation axes of the drone, the position axis ( $x$ , $y$ and $z$ ) are in relation to the window, “Yaw” angle is the horizontal angle of the center of the window related to the front of the drone, “pitch” and “roll” are angles of the drone itself (calculated from the inertial sensors only). . . . .	19
7	Attitude and position controller consisting of a two level, cascaded, structure. . . . .	20
8	Free body diagram (force diagram) of the drone $x - y$ or $z - y$ plane forces, this diagram demonstrate the relation between $x$ or $z$ acceleration ( $a_x$ or $a_z$ ) and the corresponding tilt angle. $\alpha$ represent the tilt angle (roll or pitch) and $a$ represent the acceleration in the corresponding direction. We assume that the altitude controller work as expected and the vertical speed is close to zero ( $a_y = 0$ ), and we get $ma = \arctan(\alpha)mg$ , that is, the acceleration is $a = \arctan(\alpha)g$ we use tis relation for the controller design (Section 5.2) and we also use this calculation as part of the system model in the Observer (Section 5.1) . . . . .	21

9	This figure shows in the bottom plot the position of the quadrotor during the flight time and in the top plot the $ \tilde{y}_{k k} $ 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. . . . .	23
10	The main Automata we used for the requirements specifications in the experiments. $A_{err}$ uses the <i>measurement post-fit residual</i> ( $\tilde{y}_{k k}$ ), 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 $A_x$ . . . . .	24
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.	26
12	This figure shows the drone hovering in front of the window during one of the experiments in our lab, window in the test is just 4 light balls. . . . .	27
13	The quad-rotor used for testing. . . . .	27
14	This figure present the graphics processing unit (GPU) configuration we used for image resizing. . . . .	28
15	This figure shows the four light balls we used to represent the window during the experiment. . . . .	29
16	Example of a game for vision based sensor component of a robot. The robot must maintain position of $y \leq 1$ where $H$ is high accuracy and $L$ is low accuracy position measuring tasks, and $g1$ refer to $y \leq 0.5$ , $g2$ refer to $0.5 < y \leq 1$ and $g3$ refer to $1 < y$ . . . . .	30
17	Example of a game for resource component that do not allows to exceed a time slot. In this example $T = \{t_1, t_2, t_3\}$ , the execution time of each task $t_i$ is half of a single time slot, therefore it is not possible to execute more than two tasks at the same time slot, but there is no limitation on the environment ( $\vec{y}$ ). Square state is an environment player state and round state belong to the scheduler. . . . .	32



# 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., [8, 12, 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 disciplines 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, 7, 9, 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 reactive scheduling where the schedule is directed by 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).

The idea of using reactive schedulers, as described in the preceding paragraph, was first introduced by Merav Bukra in her master thesis [6]. Merav's focus was on the algorithmic issues involved with the composition of components. The focus of this thesis is on the integration of the approach with control engineering and with its implementation in the context of a case-study in vision based control. Merav showed that we can provide tool support for the combination of components (even in a plug-and-play fashion). This thesis provides methods for control engineers for designing controllers that make use of reactive schedulers and a proof-of-concept implementation of such controllers in the context of a real case-study. Our goal in the future is to combine both theses into a tool suite where control engineers can plug their reactive components.

## 2 Problem Statement

In this thesis we focus on feedback control loop based systems as shown in Figure 1. We assume a closed (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 previous 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  $x_{k-1}$ ,  $B$  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  $Q_k$ .

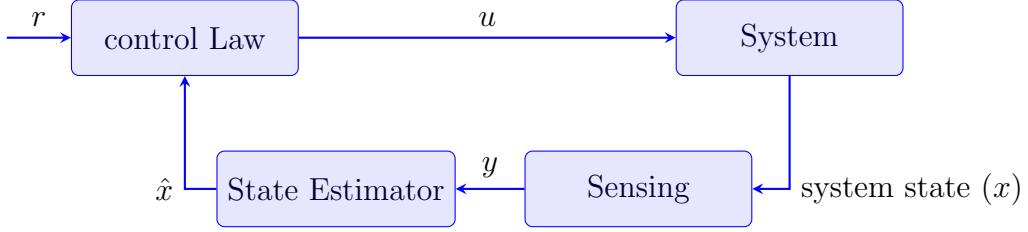


Figure 1: A typical feedback control loop where the physical plant (*System*) is monitored via an array of sensors (*Sensing*) which produce noisy sample of the state variables  $y$ . And after sensing, *State Estimator* aggregates the measurements from all the sensors and produce estimation of the current 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 ( $r$ ).

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 described 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 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 automa-

ton. 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) [10] 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 ( $P = \text{var}(\hat{x} - x)$ ) 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.

### 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 specifica-

Explain that we are proposing to focus on sensors (and actuators). The separation principle. Alignment with the structure of the ArduPilot and other controller codes.

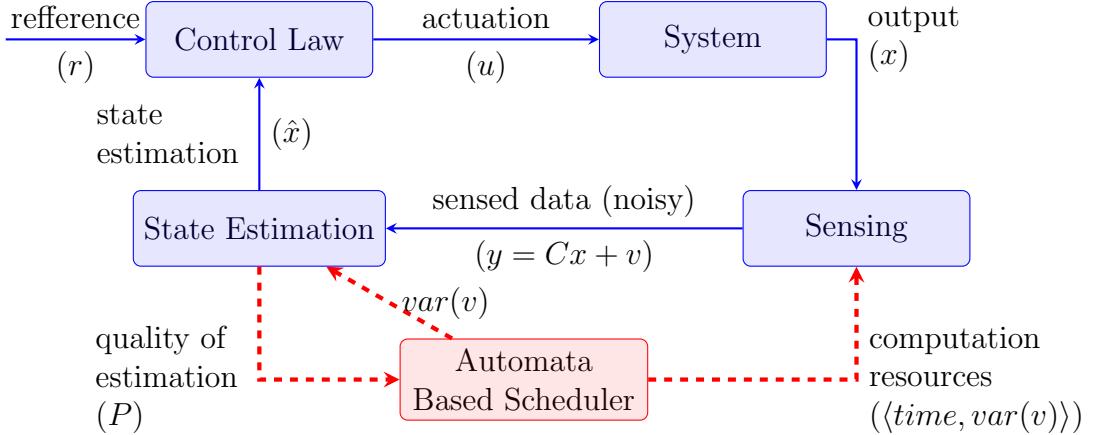


Figure 2: The controller framework we implement. Each control loop (depicted in blue) informs the resource allocator (Scheduler) of its quality of estimation, the *scheduler* will allocate CPU time accordingly in order to maintain valid estimation quality. Specifically for the Sensing processes, for 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 allocation is noted by  $\langle time, var(v) \rangle$ .

tion 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 5), we chose the ArduPilot Mega (APM) platform for experiments [10]. 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

---

```

/*
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

with very little addition to the complexity of the scheduler, as we will demonstrate in Section 4 and in Section 5 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 actuation 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.

maybe add paragraph with details and examples in the real word: changing in daylighth surface texture...

we can show difrent prespectives: 1. offline scheduling using only  $P$  (eror covariance matrix), 2. online scheduling using  $\tilde{y_k}|k$  with the explenation that in the real world the errors are not always normal.

## 4 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 im-

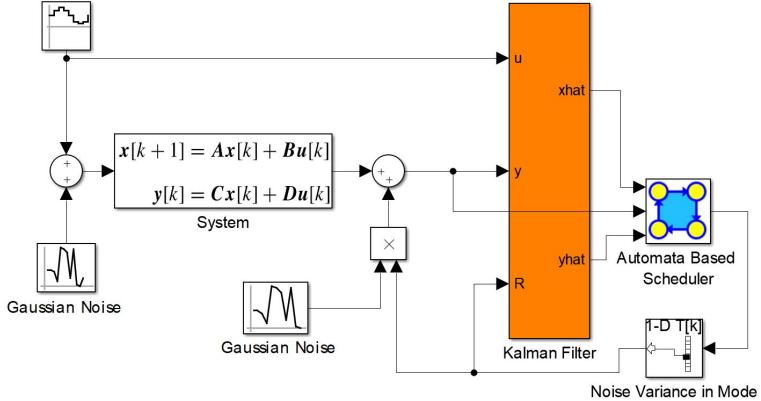


Figure 4: A Simulink model used to demonstrating and examine the proposed approach.

plementation 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.

As said in Section 3, we propose to implement the resource scheduling decisions using automata that control which procedures are invoked in the control loops. More specifically, for sensors or actuators that require heavy computations, such as vision based sensors, we propose that the software engineers develop several modes of sensing, each consumes a different amount of CPU and provides a different level of 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 4. The inaccuracies 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 4. 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 ( $u$ ), measurements ( $y$ ), and the covariance matrix of the measurement noise ( $R$ ). We assume that the measurement 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 4, the covariance matrix of the disturbance, which in this case is  $1 \times 1$  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

not realy like  
the actuators  
here, they are  
not really mod-  
eled. we con-  
sider they error  
as “process er-  
ror”

a Gaussian noise with variance  $v$ ).

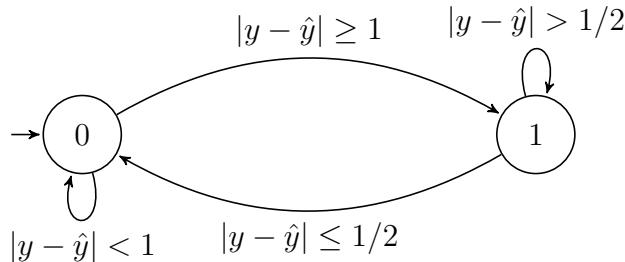
We ran the model depicted in Figure 4 with the linear time invariant system

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

taken from [mathworks.com/help/control/examples/kalman-filter-design.html](http://mathworks.com/help/control/examples/kalman-filter-design.html). As seen in Figure 4, 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 that produces normally distributed measurement errors with higher 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 states of the automaton represent operation mode of the sensor, 0 for ‘Low’ mode and 1 for ‘High’ mode.  $|y - \hat{y}|$  means how far is the last measurement from the expected measurement this value act as indication (and consider approximation) of the overall real noises, process, measurement and actuation noises together. The automaton try to formalize the expression: “use ‘High’ mode only after observing high distributed”.

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

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

Table 1: Simulation results.

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  $x$  (the real state) and  $\hat{x}$  (the Kalman filter estimated state), 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.

## 5 Case Study: Stabilizing a Quad-rotor in Front of a Findow

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](http://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<sup>1</sup>. Its goal is to regulate the position of the quadrotor by tilting it. Note that rotations of the quadrotor by a small angle 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.

---

<sup>1</sup>In the experiment, to simplify the image processing algorithm, we marked the corners of the window with led lights.

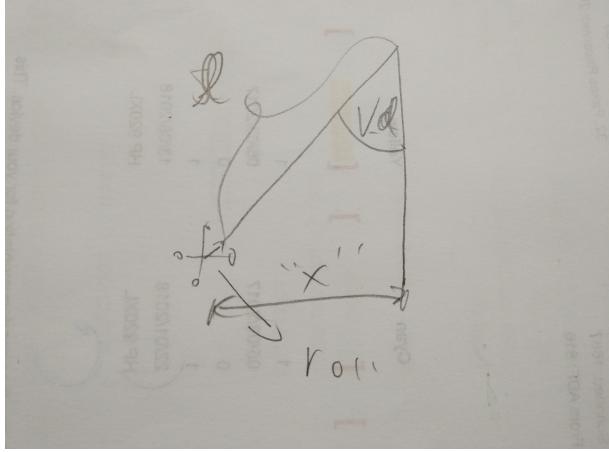


Figure 5: Geometric relation between  $\alpha$  (the angle derived from  $V_d$ )  $l$  (distance from window extracted from  $sz$ ) and the position of the drone among the  $x$  axis. (TODO - need to replace image)

## 5.1 Observer Design

We first implemented an observer based on the work of Efraim [11]. The observer gets the positions of the widow corners from the image (see Section 7 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 yaw angle and the altitude of the drone, respectively. Means the drone is always facing the center of the window and at the altitude of the center of the window. **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 <sup>2</sup>. **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 measure the angular position of the drone in relation to the window (parallel to the ground) and used to control the position of the drone among the  $x$  axis. Note that although  $V_d$  is not directly measure the linear  $x$  position of the drone, we can estimate the linear  $x$  position by simple geometry: as show in Figure 5  $x = l * \sin(V_d)$  were  $l$  (distance from window) can be calculated from  $sz$ . In our case because  $V_d$  is usually relatively small we approximate it by  $x = l * V_d$ .

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 4, 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

don't know if some of this need to be moved to Section 12.1

<sup>2</sup>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.

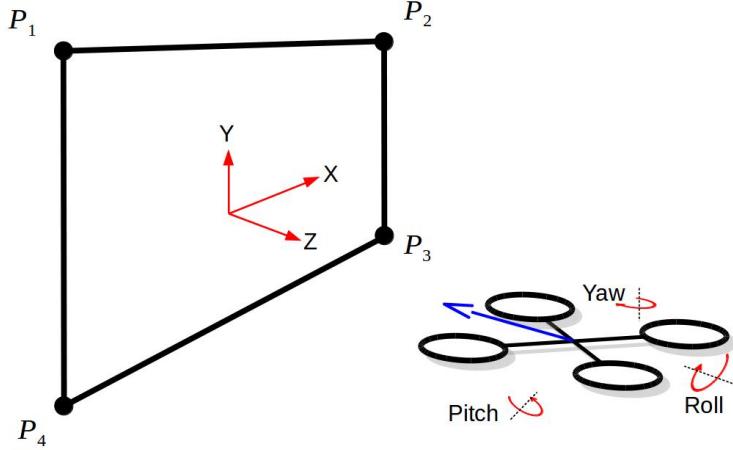


Figure 6: The coordinate system and the rotation axes of the drone, the position axis ( $x$ ,  $y$  and  $z$ ) are in relation to the window, “Yaw” angle is the horizontal angle of the center of the window related to the front of the drone, “pitch” and “roll” are angles of the drone itself (calculated from the inertial sensors only).

complexity in the code, we 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 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 ( $C^{-1}y_k$ ) <sup>3</sup>:  $\hat{x}_{k|k} = K\hat{x}_{k|k-1} + (1-K)C^{-1}y_k$  were  $K$  is constant that equals the Kalman filter gain ( $K_k$ ) at a steady-state, see Section 12.1 for detail information.

need to add section on the system linearized model?

The image processing algorithms can run in different operation modes, each mode with different accuracy (see Section 5.3). 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.

## 5.2 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 6) 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 desire reference position and the coordinates are according to Figure 6.

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

---

<sup>3</sup>The matrix  $C$  is the measurement matrix as shown in the model at Figure 4

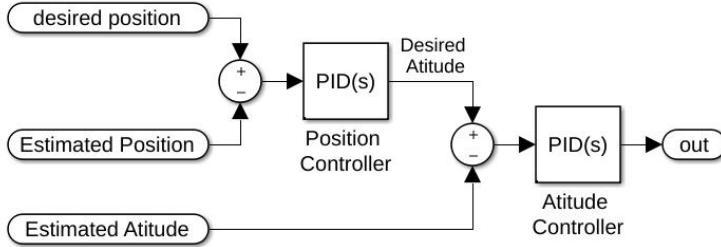


Figure 7: Attitude and position controller consisting of a two level, cascaded, structure.

as input the required pitch or roll angle, 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 7. Altitude, yaw and  $x$  and  $z$  position is regulated relative to the window. The inertial pitch and roll feedback for the low level controllers is related to the drone body and is generated by the existing *Attitude and Heading Reference system* (AHRS) library of ArduPilot (see Section 7) and the position feedback ( $x$ ,  $y$ , and  $z$  position) came directly from the observer described in Section 5.1.

The position controller is as follows: The altitude ( $y$  position) is obtained directly from the  $S_y$  value explained in Section 5.1 and it is regulated by changing the overall “throttle” of the rotors, this is good approach assuming the drone pitch and roll angles are close to zero then  $S_y$  is almost proportional to the real altitude of the drone and the trust vector is pointed up. The  $z$  position is assumed to be the distance from the center of the window and is calculated as proportional to the window size  $sz$  value, it is regulated by changing the pitch angle. Assuming the drone is always facing the center of the window (yaw angle is zero) the pitch angle produce acceleration to the pitch direction that is to the window direction or from the window meaning its will decrease or increase the  $z$  position. The  $x$  position (displacement) is regulated by changing the roll angle. Similar to  $z$  position, assuming the drone is always facing the center of the window the roll angle produce acceleration that increase or decrease the  $x$  position depending on the roll angle direction.

As demonstrated in Figure 8, the  $x$  acceleration is equal to  $a_x = g \arctan(\alpha_{roll})$  and  $x$  acceleration is equal to  $a_z = g \arctan(\alpha_{pitch})$ . In practice, during this work we usually assume linear proportional acceleration meaning  $a_x = g \cdot \alpha_{roll}$  and  $a_z = g \cdot \alpha_{pitch}$ .

Note that if  $x$  is not zero then the drone is not parallel to the window and the roll direction (that used to regulate  $x$  position) is also affects the  $z$  position. Similarly, if the drone is not parallel to the window and the pitch direction (that used to regulate  $z$  position) is also affects the  $x$  position. Although in theory  $x$  controller interrupt the  $z$  controller and vice versa, in practice even if the drone displacement is relatively large,

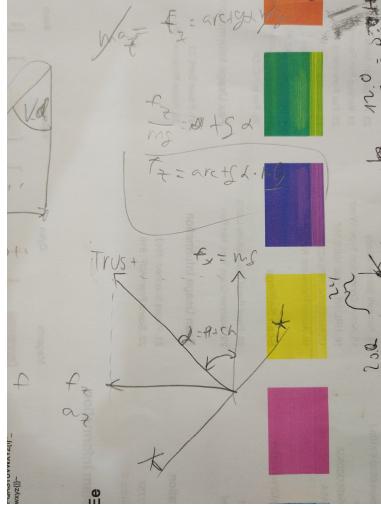


Figure 8: Free body diagram (force diagram) of the drone  $x - y$  or  $z - y$  plane forces, this diagram demonstrate the relation between  $x$  or  $z$  acceleration ( $a_x$  or  $a_z$ ) and the corresponding tilt angle.  $\alpha$  represent the tilt angle (roll or pitch) and  $a$  represent the acceleration in the corresponding direction. We assume that the altitude controller work as expected and the vertical speed is close to zero ( $a_y = 0$ ), and we get  $ma = \arctan(\alpha)mg$ , that is, the acceleration is  $a = \arctan(\alpha)g$  we use this relation for the controller design (Section 5.2) and we also use this calculation as part of the system model in the Observer (Section 5.1)

when all the controllers executed together they constantly fix those side-affects and because the  $x$  position is generally decreased during this process we eventually converge to a stable state where the drone is near-parallel to the window.

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 highest resolution observer.

### 5.3 Analysis and Specification Automata

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 6). Our goal is to achieve maximum performance with minimal amount of processing time. Obviously, both goals cannot be achieved together. We therefore 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 5.1). For simplicity, we consider in this thesis 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  $9.5\text{cm}$  (in the  $x$  axis) with the cost of 30% CPU usage. On the other hand, the low quality mode provides mean error of  $30\text{cm}$  (three times the error obtained in the high quality mode) using only 2.1% CPU usage<sup>4</sup>.

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

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  $L$  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 6 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 6 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

---

<sup>4</sup>The CPU usage percentage is the average: (total time spend in image processing)/(total flight time).

there are 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.

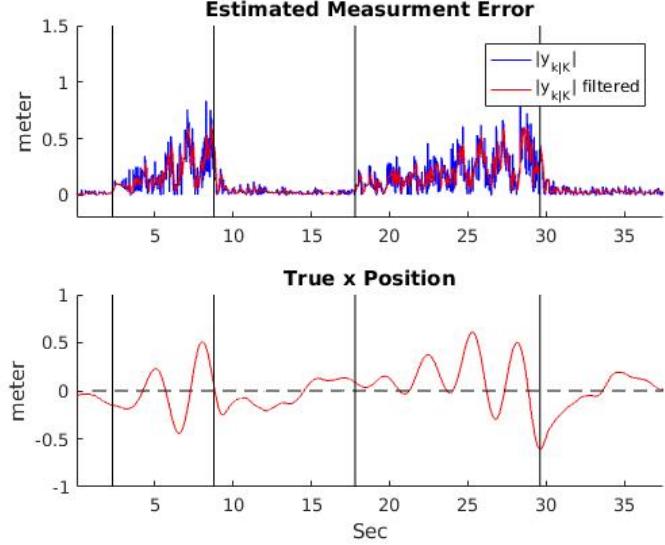


Figure 9: This figure shows in the bottom plot the position of the quadrotor during the flight time and in the top plot the  $|\tilde{y}_{k|k}|$  value during the flight. During the flight we manually switched between low and high resolution for further analysis. The modes (high or low) are separated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

## 6 Results

Table 2 summarizes the result of the flight experiments made using the specification automata described in Section 5.3. 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 7) and data from the software itself.

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 described in Section 5.3 with the threshold values of 10, 20 and 30cm respectively. Those

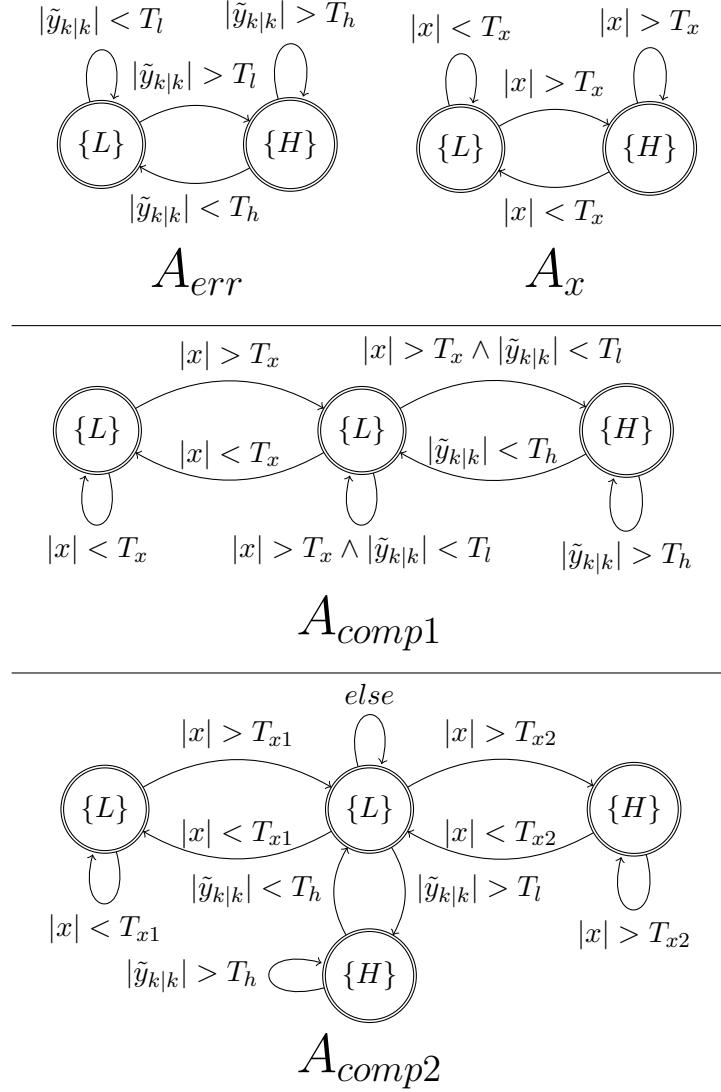


Figure 10: The main Automata we used for the requirements specifications in the experiments.  $A_{err}$  uses the *measurement post-fit residual* ( $\tilde{y}_{k|k}$ ), if  $|\tilde{y}_{k|k}|$  is too small then the next iteration will use small resolution image ( $L$ ), and if it is too large the high 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  $A_x$ .

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 reactivity. The scheduling scheme we are proposing allows the system to adapt itself to changes in environmental conditions, demonstrated as follows. We experimented in exposing the flying drone to time-varying 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.

## 7 Experiment Setup

### 7.1 Hardware - the Quad-Rotor

In order to perform the experiments we used an off-the-shelf quadrotor shown in Figure 13 with improved flight control: a *raspberry-pi*, a single-board computer [22], with the *navio2* [13] shield. *Navio2* is extension board (shield) for the *Raspberry pi* platform, it provides the necessary interfaces such as remote control inputs and motors output, and it is equipped with on-board Micro Electro-Mechanical System (MEMS) sensors, such as rate gyros and accelerometers. We implemented the controllers and the schedulers as libraries

Table 2: Test Results

	Schedule	% CPU	$\text{mean}( x )$ (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

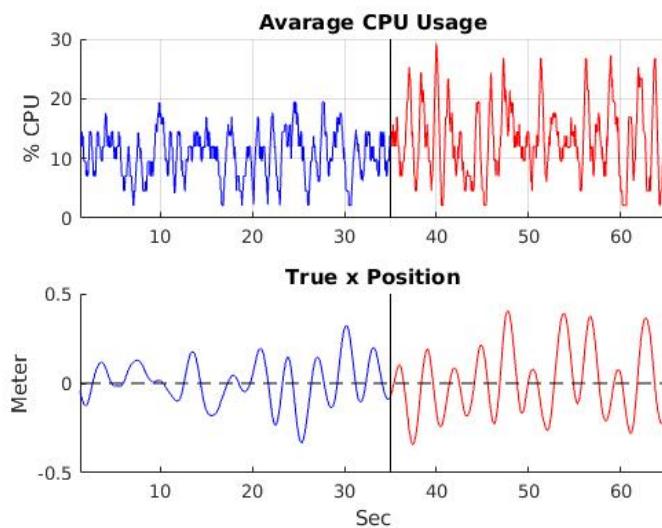


Figure 11: This figure demonstrates how the reactive scheduler adapt to the environment changes, in particular, to change of the wind. The upper part shows the average CPU usage during the flight, and the bottom is the true position at the same time. The first part of the flight (the left Blue graph) 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.



Figure 12: This figure shows the drone hovering in front of the window during one of the experiments in our lab, window in the test is just 4 light balls.



Figure 13: The quad-rotor used for testing.

for the ArduPilot Mega (APM) autopilot software [10].

## 7.2 Vision Based Sensor

As explained in Section 5 we use vision-based sensor for identify the window. For this task we used the standard raspberry-pi camera module (V1) (see [www.raspberrypi.org/documentation/hardware/camera/](http://www.raspberrypi.org/documentation/hardware/camera/)) mounted in the front of the drone. This camera can take pictures in different resolutions and at different frame rate. The controller software implemented in c++ therefore we initially use the most common used driver for C++ called *RaspiCam* (see [www.uco.es/investiga/grupos/ava/node/40](http://www.uco.es/investiga/grupos/ava/node/40)). RaspiCam have two main difficulties for us:

1. The raspberry-pi camera take a picture in a constant pre-define frame rate <sup>5</sup>, this mean that if we want to grab an image the driver (RaspiCam) will wait till the next frame is taking. This approach is not appropriate for real-time system.
2. As describe in Section 5 we need to switch image resolution during the flight many times, raspberry-pi camera resolution is defined only at initialization, therefore

---

<sup>5</sup>The constant pre-define frame rate is only wen using “vid” mod which allows fast frame rate

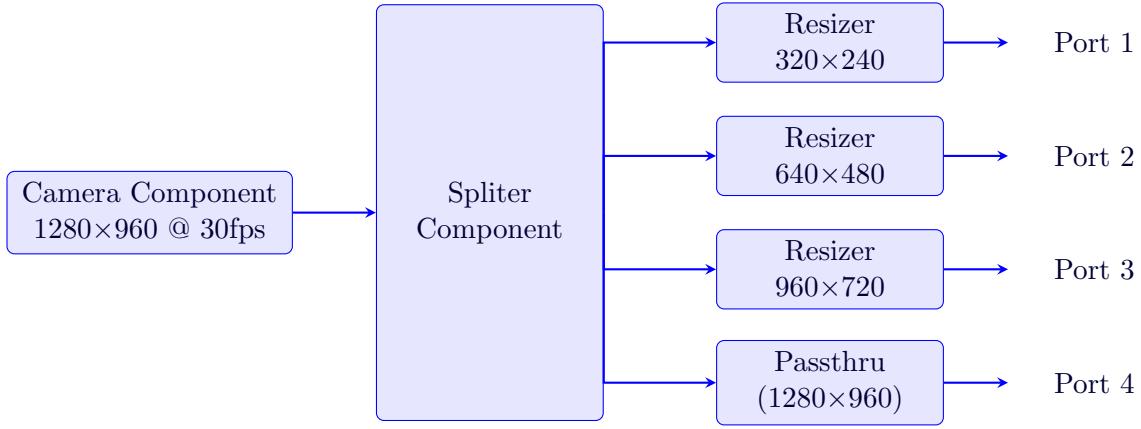


Figure 14: This figure present the graphics processing unit (GPU) configuration we used for image resizing.

changing resolution of the camera results in significant delay till the camera is ready at the new resolution.

The first difficulty has solved by modifying RaspiCam driver, we add to RaspiCam the ability to notify the client when the next frame is ready, this way whenever a new frame is taken the callback function is called. We also add this modification to the official RaspiCam code that is maintained in [github.com/cedricve/raspicam](https://github.com/cedricve/raspicam).

In order to overcome the second difficulty we try to take pictures at the high resolution and then resize by software (using OpenCV) to the desire resolution, this solution still add undesirable delay for each frame that is taking. At the end, we create a new camera driver based on the work of Chris Cummings at [robotblogging.blogspot.co.il/2013/10/an-efficient-and-simple-c-api-for.html](http://robotblogging.blogspot.co.il/2013/10/an-efficient-and-simple-c-api-for.html). In this driver we use the internal GPU hardware of the Raspberry-py in order to resize the frame by hardware. Using the ARM side libraries *userland* <https://github.com/raspberrypi/userland> we construct a up to 4 video ports each one from different resolution resizer all connected to the camera port (see Figure 14), now each time we want to take image we can choose the image resolution by retrieving the image from the corresponded port.

In the case-study, shown in Section 5, we always use the camera at 30 frames per second. In order to simplify the image processing algorithm, we marked the four corners of the window with four illuminated balls (see Figure 15). 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” but require more processing time.

### 7.3 Reference Measurement with OptiTrack System

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](http://optitrack.com)) system

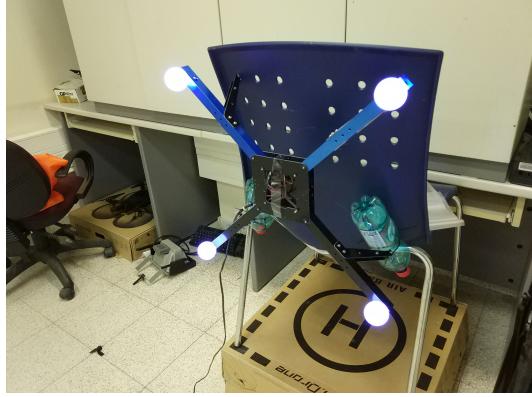


Figure 15: This figure shows the four light balls we used to represent the window during the experiment.

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 *OptiTrack* system is sent over the client-server data streaming protocol *NatNet*. For best synchronization, we create a *NatNet* client library inside the APM code. This library allows the controller check anytime for the current state and log that information in parallel to the other controller parameters. We also use the OptiTrack information in early stages of the research as the feedback of the controller before we use the camera .

## 8 Scheduler Interface Prototype

I dont know if  
this section is  
too much (is  
mostly taken  
from the pro-  
posal)

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. Specifying resource requirements using periods has advantages due to simplicity and analyzability, but has limited expressiveness, as elaborated in [2].

In Section 3 we have suggested to use automata for describe the task specification for the scheduler. In this section we elaborate the technical details of the specification automata, techniques and tools that can be used to analyze the system, and in the last part of this section we show a working scheduler prototype that we implement to create to prove the concept.

fix title

### 8.1 A Schedule Component

**Definition 2.** A component is a set,  $T$ , of subroutines (representing, e.g., a class in C++) with a generalized Büchi game (GNBG),  $G = \langle A, \langle P_{schd}, P_{env} \rangle \rangle$ , over a generalized Büchi automaton  $A = \langle Q, \Sigma, \Delta, q_0, \mathcal{F} \rangle$ , such that:

1. The alphabet is  $\Sigma = 2^T \cup \mathbb{R}^n$  where  $n$  is the number of scheduler feedback variables in the system (real numbers that the scheduler can base its decision on).

Add an ap-  
pendix on  
games

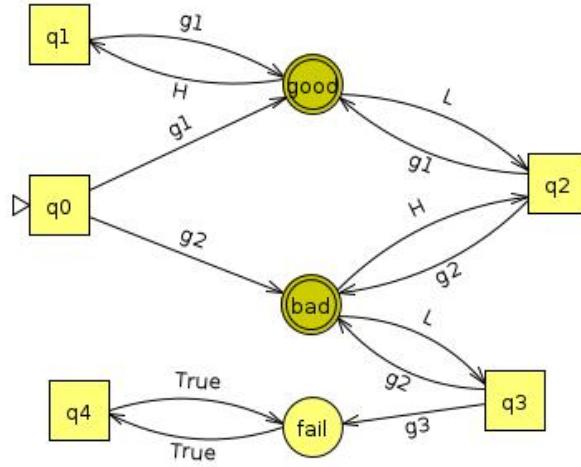


Figure 16: Example of a game for vision based sensor component of a robot. The robot must maintain position of  $y \leq 1$  where  $H$  is high accuracy and  $L$  is low accuracy position measuring tasks, and  $g_1$  refer to  $y \leq 0.5$ ,  $g_2$  refer to  $0.5 < y \leq 1$  and  $g_3$  refer to  $1 < y$

2. *Alternating turns with different parts of the alphabet: The transitions of  $A$  are of the form:  $s_{schd} \xrightarrow{\sigma_T} s_{env}$  where  $s_{schd} \in P_{schd}$ ,  $s_{env} \in P_{env}$ , and  $\sigma_T \subseteq T$  (called “scheduler” transitions ), or of the form  $s_{env} \xrightarrow{\sigma_e} s_{schd}$ , where  $s_{schd} \in P_{schd}$ ,  $s_{env} \in P_{env}$ , and  $\sigma_e \in \mathbb{R}^n$  (called “environment” transitions).*
3. *The environment plays first:  $q_0 \in P_{env}$ .*

The idea is that each component is designed to handle a specific task in the system. All its subroutines serve this task. The scheduling game,  $G$ , of the component specifies the interaction between the subroutines and the environment. The acceptance condition encodes the component requirements.

An example of a component acting as the vision based sensor of a robot is demonstrated in Figure 16. The component estimates the robot position from an image. It has two operation modes: high ( $H$ ) or low ( $L$ ) accuracy. The scheduler feedback variable  $y$  is the estimated position, in this example the component requirement is to ensure that  $y$  never exceeds one (in position  $y = 1$  there is a cliff).

A scheduling game  $G$  is a scheduler-against-environment game as follows: at each scheduler state  $s_{schd} \in P_{schd}$  is a single time slot where the scheduler need to choose a transition  $s_{schd} \xrightarrow{\sigma_T} s_{env}$  to a new state  $s_{env} \in P_{env}$  and then execute all the tasks in  $\sigma_T$ , this transition lead to an environment state  $s_{env} \in P_{env}$  that represent the evolution of the system state till the next time step cause by the actuators and the environment, each transposition  $s_{env} \xrightarrow{\sigma_e} s_{schd}$  correspond to a possible new value of the system state that

this example is too early because the reader doesn't know the terms yet

equals to  $\sigma_e$ . The scheduler goal is to pass infinitely many times in acceptance state during the execution. Note, formally scheduler feedback variable is a vector  $Y \in \mathbb{R}^n$  that may includes state variables and internal variable of the components.

## 8.2 Component Games Composition

The system composed of multiple components that need to be executed simultaneously by the same scheduler. Each component introduce its own game therefore we will construct a composed component that correspond to all the system components together, then the scheduler have only one game to win for handle all the components.

**Definition 3.** *The composition of the components  $C_1 = \langle A_1, \langle P_{schd}^1, P_{env}^1 \rangle \rangle$  and  $C_2 = \langle A_2, \langle P_{schd}^2, P_{env}^2 \rangle \rangle$  is the component  $C_c = C_1 \times C_2$  defined as follows: The subroutines set is  $T_c = T_1 \cup T_2$ . The schedule game is  $G_c = G_1 \times G_2 = \langle A_1 \times A_2, \langle P_{schd}^1 \times P_{schd}^2, P_{env}^1 \times P_{env}^2 \rangle \rangle$ , that is, the automaton  $A_c = \langle Q_1 \times Q_2, \Sigma_c, \Delta_c, Q_0^1 \times Q_0^2, Acc_1 \cup Acc_2 \rangle$  is a product of  $C_1$  and  $C_2$  automata such that:*

1.  $\Sigma_c = T_c \times \mathbb{R}^n$
2. *Environment transitions: for each  $y \in \mathbb{R}^n$ ,  $s_1, s'_1 \in Q_1$  and  $s_2, s'_2 \in Q_2$  implies that:  
 $s_1 \xrightarrow{y} s'_1 \in \Delta_1$  and  $s_2 \xrightarrow{y} s'_2 \in \Delta_2$  if and only if  $\langle s_1 s_2 \rangle \xrightarrow{y} \langle s'_1 s'_2 \rangle \in \Delta_c$*
3. *Scheduler transitions: for each  $T'_1 \subseteq T_1$ ,  $T'_2 \subseteq T_2$ ,  $s_1, s'_1 \in Q_1$  and  $s_2, s'_2 \in Q_2$  implies that:  
 $s_1 \xrightarrow{T'_1} s'_1 \in \Delta_1$  and  $s_2 \xrightarrow{T'_2} s'_2 \in \Delta_2$  if and only if  $\langle s_1 s_2 \rangle \xrightarrow{T'_1 \cup T'_2} \langle s'_1 s'_2 \rangle \in \Delta_c$*

Note: for constructing the composed component  $C_c$  for all the system components, assuming the system consist of components  $C_1, \dots, C_m$  then we will compose the components one by one:  $C_c = (\dots((C_1 \cdot C_2) \cdot C_3) \cdot \dots \cdot C_m)$ .

## 8.3 Add Resources Limitation

In addition to the component requirements we also have the resources requirements, we have limited resources (limited CPU abilities) that need to be taking in account. Usually resources requirements are reflected as limited time slot that not allows tasks to exceed its size, for example, a tasks set that needs one second to execute cannot be executed in half second time slot. We will represent those limitations with a special and simple component  $C_{resource}$  that can be generated automatically as follows:

assuming we know the maximum duration of each subroutine of each component in the system (it can be result of analyzing, testing, or manually pre-defined) we construct component with the tasks set  $T_{resource}$  that is union of all the subroutines, and the game is as demonstrate in Figure 17 with one state for environment player (square state) and one

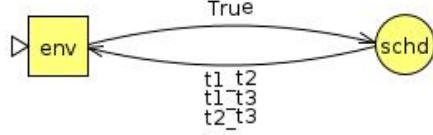


Figure 17: Example of a game for resource component that do not allows to exceed a time slot. In this example  $T = \{t_1, t_2, t_3\}$ , the execution time of each task  $t_i$  is half of a single time slot, therefore it is not possible to execute more than two tasks at the same time slot, but there is no limitation on the environment ( $\vec{y}$ ). Square state is an environment player state and round state belong to the scheduler.

for the schedule (circle state), the environment have no limitations but the scheduler can execute only tasks set that not exceeds the time slot.

Finally, the resource component is composed with all the rest components of the system result in the schedule component  $C_s = C_{reasource} \cdot C_c$ .  $C_s$  is the only component that actually will be schedule.

Make the states accepting

## 8.4 Wining the Game - Find the Strategy

After we construct the game  $C_s$  that aggregates all the system constrains we define a “wining” strategy  $A_s.strategy \in NBA$  for the scheduler.  $A_s.strategy$  is in fact a reduction of the game automaton  $A_s$  of  $C_s$  its contain all the environment transitions in  $A_s$  but only the scheduler transitions that allays lead to a winning of the scheduler, that is we visit infinitely times a state of every color  $Acc' \in Acc$ . Example of such strategy is sown in Figure 18 this is a wining strategy for the scheduler in the game presented in Figure 16 discussed before. We call this winning strategy guarded automaton .

we can add here reference to the simpler guarded automata presented in the previous sections

## 8.5 Execution of the scheduler

Each time the system is executed the scheduler just walk through the strategy automaton  $A_s$  as describe by the pseudo code 1. The time is divided into segments of equal length called “time step”. At the beginning of each time step the current automaton state is environment player state, so we pick the

continue verbal explanation of the pseudo cod

## 8.6 Correctness

**Definition 4.** A component  $C_i$  is “complete” if for each  $t_1, \dots, t_m \subseteq T_i$ ,  $\vec{y}_1, \dots, \vec{y}_m \in \mathbb{R}^n$  and  $q_e \in P_{env}$  such that there is a path from  $q_0$  to  $q_e$  with the word  $w = \vec{y}_1, t_1, \vec{y}_2, t_2, \dots, \vec{y}_m, t_m$

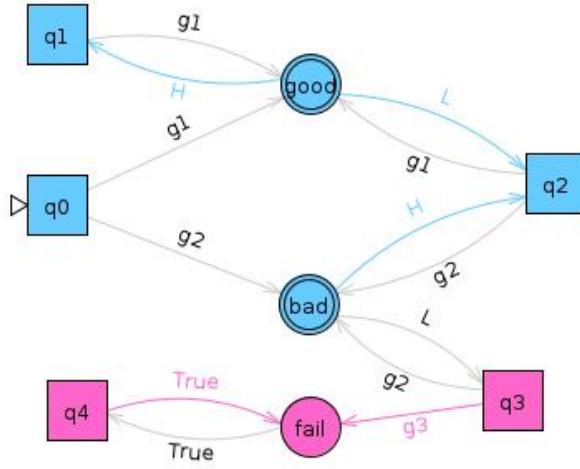


Figure 18: A strategy solution for the game presented in Figure 16. Blue states are all the states which the scheduler have a winning strategy from them, blue transitions are the needed scheduler reaction to win the game. Note: if the initial state is blue then the system is schedulable.

we have that for each  $\vec{y}_{m+1}$  that can be observed after any run of the system that corresponds to this word (i.e., after each observation  $\vec{y}_j$  we schedule  $t_j$ ) there is a transition from  $q_e$  labeled with  $\vec{y}_{m+1}$ .

maybe stability requirements

**Definition 5.** A complete component  $C_i$  is valid if in any execution whose observations and task schedules constitute a word  $\omega \in \mathcal{L}(A_i)$  the component achieves its control objectives.

**Claim 1.** Assuming  $C_1$  and  $C_2$  are a valid components then the composed component  $C_c = C_1 \cdot C_2$  is also a valid component.

maybe just the transitions rule is not trivial

*Proof.* 1.  $C_c$  is a component: trivial

2.  $C_c$  is complete component: Assume that after we get to state  $\langle q_1, q_2 \rangle \in P_{env}^1 \times P_{env}^2$  we may observe  $\vec{y}$ , then because  $C_1$  is complete there is a transition  $q_1 \xrightarrow{\vec{y}} q'_1$  and same for  $q_2 \xrightarrow{\vec{y}} q'_2$  in  $C_2$ , therefore by the composition definition  $A_c$  contains the transition  $\langle q_1, q_2 \rangle \xrightarrow{\vec{y}} \langle q'_1, q'_2 \rangle$

3.  $C_c$  satisfy its requirements: by composition definition the word  $\omega = \vec{y}_1, t_1, \vec{y}_2, t_2, \dots \in \mathcal{L}(A_c)$  if and only if there exist words  $\omega_1 = \vec{y}_1, t_1^1, \vec{y}_2^1, t_2, \dots \in \mathcal{L}(A_1)$  and  $\omega_2 = \vec{y}_1, t_1^2, \vec{y}_2^2, t_2^2, \dots \in \mathcal{L}(A_2)$  such that  $t_i = t_i^1 \cup t_i^2$ . Therefore, for each  $\omega = \vec{y}_1, t_1, \vec{y}_2, t_2, \dots \in \mathcal{L}(A_c)$  a system execution that before each time slot  $j$  observe  $\vec{y}_j$  and then execute the tasks  $t_i$  in that slot is in particular execute  $t_i^1$  and because  $\omega_1 = \vec{y}_1, t_1^1, \vec{y}_2^1, t_2, \dots \in$

---

**Algorithm 1** Scheduler run-time operation

---

```

1: procedure S CHEDULE( $A_s$ )       $\triangleright$  schedule the system with strategy automaton  $A_s$ 
2:   VALIDATE( $A_s$ )                   $\triangleright$  validate that  $A_s$  is wining strategy
3:    $cur\_state \leftarrow \text{INITIAL}(A_s)$            $\triangleright$  start with the initial state

4:   loop
5:      $\vec{y} \leftarrow \text{GETSTATEVARS}()$ 
6:      $cur\_state \leftarrow \text{NEXT}(cur\_state, \vec{y})$        $\triangleright$  take environment transition for  $\vec{y}$ 
7:     if  $cur\_state == \text{None}$  then
8:       Exit                                 $\triangleright$  unexpected  $\vec{y}$  value, fail to proceed
9:     end if

10:     $\delta \leftarrow \text{GETFIRSTTRANSITION}(cur\_state)$        $\triangleright$  take scheduler transition
11:     $Tasks \leftarrow \text{TASKS}(\delta)$ 
12:     $cur\_state \leftarrow \text{TOSTATE}(\delta)$ 
13:    for all  $t \in Tasks$  do       $\triangleright$  execute all the tasks defined by the transition
14:      EXECUTE( $t$ )
15:    end for

16:    Wait till next time step
17:  end loop
18: end procedure

```

---

$L(A_1)$  satisfies  $C_1$  requirements  $\omega$  also satisfying  $C_1$  requirements, similarly,  $\omega$  also satisfying  $C_2$ , then  $\omega$  is a system execution that satisfies the requirement from the component (for the system).

□

**Definition 6.** A system is “schedulable” if for any possible change of environment there is an infinite schedule execution that satisfy the system requirements.

**Claim 2.** Assuming  $C_1, C_2, \dots, C_m$  are a valid components then the composed component  $C_s = C_1 \cdot C_2 \times \dots \times C_m \cdot C_{\text{resource}}$  is valid component and if there is a wining strategy for the scheduler  $A_s$  then the system defined by  $C_s$  is schedulable system.

*Proof.* First note that  $C_{\text{resource}}$  is valid component: The only environment state of  $A_{\text{resource}}$  accept all  $\mathbb{R}^n$  for the observations therefore  $C_{\text{resource}}$  is complete, the only requirement from  $C_{\text{resource}}$  is that the time step is never exceeded its length, this is guaranteed by the contraction of scheduler transitions then  $C_{\text{resource}}$  is also satisfy its requirement so  $C_{\text{resource}}$  is valid component. All components  $C_1, C_2, \dots, C_m$  and  $C_{\text{resource}}$  are valid components then by claim 1  $C_s$  is valid component.

Assume there is a wining strategy for  $C_s$  then the scheduler that random walk through the wining strategy automaton, and operate as elaborate in algorithm 1 is:

1. Never get stuck:  $C_s$  is complete so each environment state of  $A_s$  have transition for any possible observation and any strategy of scheduler must keep all environment

transitions therefore the execution never get stuck at environment state, and each scheduler state in the strategy have at least one outcome transition because if this is not the case then the strategy is not a winning strategy.

2. The component  $C_{resource}$  game have only transitions with task sets that even at the worst case they never exceeds a single time step and  $T_{resource}$  contains all the system components tasks, therefore,  $C_s$  also do not contain tasks sets that exceeds a single time step, therefore a schedule of a longer tasks set is not possible in algorithm 1.
3. Satisfy the system requirements: The component  $C_s$  satisfy its requirements, and the random walk of algorithm 1 is also constructing a word  $\omega \in L(C_s)$ , therefore by the definition of component that "satisfy its requirement", algorithm 1 satisfy all the components requirements.

□

**Remark 1.** *The fact that a system is schedulable, is not necessarily implies the there is a winning strategy.*

## 9 Few Words about Implementation

## 10 Related Work

This thesis is mainly based on the RTComposer paper [2] and the followed work of Merav [6]. In this thesis as in RTComposer we present a framework for design cyber-physical systems with dynamic scheduling for real-time. The RTComposer the framework is component-based in the sense that a single component correspond to a single control loop in a complex cyber-physical system. Each component has a clearly specified interface that includes method for sensing, method for actuation, and additional computational methods. Each component have its requirement specification given by finite automaton over infinite words, every infinite sequence over the automaton specifies, for every time slot, the set of the component computational methods to invoke in that slot.

In this thesis we also propose to use automata based scheduler as automata specification is expressive, analyzable, and composable specification framework as stated in section 3. In this work unlike RTComposer we prefers to loosen the definition of a component and remove the separation of sensing and actuation methods ...

## 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 Appendix

### 12.1 Kalman Filter and State Estimation

The 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 (which 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  $\alpha$  is significant high,  $f_l$  is considered *low-pass*, as rapid value changes less significant and 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 Priori 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:

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

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 Complementary-Kalman design (with physical computation)

## 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] R. Alur and G. Weiss. RTComposer: A framework for real-time components with scheduling interfaces. *8th ACM/IEEE Conference on Embedded Software*, 2008.
- [3] K.-E. Arzén, A. Cervin, J. Eker, and L. 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] K. J. Åström and T. Hägglund. *Advanced PID control*. ISA-The Instrumentation, Systems and Automation Society, 2006.

- [5] J. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. T. Rajan, H. Roeck, and R. 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] M. Bukra. GameComposer: A Framework for Dynamic Scheduling. Master’s thesis, Ben-Gurion University, Israel, October 2014.
- [7] A. Chakrabarti, L. de Alfaro, T. Henzinger, and M. Stoelinga. Resource interfaces. In *Embedded Software, 3rd International Conference*, LNCS 2855, pages 117–133, 2003.
- [8] A. K. Das, R. Fierro, V. Kumar, J. P. Ostrowski, J. Spletzer, and C. J. Taylor. A vision-based formation control framework. *IEEE transactions on robotics and automation*, 18(5):813–825, 2002.
- [9] L. de Alfaro and T. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.
- [10] A. Developers. ArduPilot Mega (APM): open source controller for quad-copters. [www.arducopter.co.uk](http://www.arducopter.co.uk).
- [11] H. Efraim. *Vision Based Control of Micro Aerial Vehicles in Indoor Environments*. PhD thesis, Ben Gurion University of The Negev, 2017.
- [12] H. Efraim, S. Arogeti, A. Shapiro, and G. Weiss. Vision based output feedback control of micro aerial vehicles in indoor environments. *Journal of Intelligent & Robotic Systems*, 87(1):169–186, Jul 2017.
- [13] EMLID. navio: Linux autopilot on Raspberry Pi. <https://emlid.com/introducing-navio2/>.
- [14] M. Esnaashari and M. 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. 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] W. T. Higgins. A comparison of complementary and kalman filtering. *IEEE Transactions on Aerospace and Electronic Systems*, (3):321–325, 1975.
- [17] R. E. 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. Lee. What's ahead for embedded software. *IEEE Computer*, pages 18–26, September 2000.
- [19] E. 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] J. Liu, N. Ozay, U. Topcu, and R. M. Murray. Synthesis of reactive switching protocols from temporal logic specifications. *IEEE Transactions on Automatic Control*, 58(7):1771–1785, 2013.
- [21] A. Mok and A. 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] Raspberry Pi Foundation. Raspberry pi. [www.raspberrypi.org/products/raspberry-pi-3-model-b/](http://www.raspberrypi.org/products/raspberry-pi-3-model-b/).
- [23] J. Regehr and J. 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. Sangiovanni-Vincentelli, L. Carloni, F. D. 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] O. Shakernia, Y. Ma, T. J. Koo, and S. Sastry. Landing an unmanned air vehicle: Vision based motion estimation and nonlinear control. *Asian journal of control*, 1(3):128–145, 1999.
- [27] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *Trans. on Embedded Computing Sys.*, 7(3):1–39, 2008.
- [28] P. 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] G. Weiss and R. Alur. Automata based interfaces for control and scheduling. In *International Workshop on Hybrid Systems: Computation and Control*, pages 601–613. Springer, 2007.



Need to update the hebrew pages