A

AdaFT: A Framework for Adaptive Fault Tolerance for Cyber-Physical Systems

YE XU, University of Massachusetts Amherst
ISRAEL KOREN, University of Massachusetts Amherst
C. MANI KRISHNA, University of Massachusetts Amherst

Cyber-physical systems frequently have to use massive redundancy to meet application requirements for high reliability. While such redundancy is required, it can be activated adaptively, based on the current state of the controlled plant. Most of the time the plant is in a state that allows for a lower level of fault-tolerance. Avoiding the continuous deployment of massive fault tolerance will greatly reduce the workload of the CPS, and lower the operating temperature of the cyber sub-system, thus increasing its reliability. In this paper, we extend our prior research by demonstrating a software simulation framework (AdaFT) that can automatically generate the *sub-spaces* within which our adaptive fault tolerance can be applied. We also show the *theoretical benefits* of AdaFT, and its *actual implementation* in several real world CPSs.

CCS Concepts:•**Computer systems organization** → **Embedded systems;** *Redundancy;* Robotics;

Additional Key Words and Phrases: Fault Tolerance, Cyber-Physical System, Processor Thermal Reliability

ACM Reference Format:
Ye Xu, Israel Koren, C. M. Krishna. AdaFT: A Framework for Adaptive Fault Tolerance for Cyber-Physical Systems. *ACM Trans. Embedd. Comput. Syst.* V, N, Article A (January YYYY), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Dramatic changes have occurred over the past few years in cyber-physical systems (CPS). Such systems range in complexity from simple and small-scale to extremely complex, large-scale systems. The traditional approach to controlling CPSs has been to use a large number of microcontrollers, each dedicated to performing a certain subset of the computational tasks, and interacting with one another. For example, an automotive application might have a microcontroller entirely dedicated to braking control, and another dedicated to cruise control. Yet another subset may be dedicated to managing the entertainment system.

More recently, in an effort to provide increased reliability and reduce costs, designers have been turning to a more flexible approach, with a shared, integrated, computational platform. Such a platform is responsible for the totality of the control activity; individual cores may be shared by different functions. The same computer platform can run widely varying tasks, whose importance may range from non-critical to life-critical. Tasks can be remapped from one processor to another depending on prevailing load conditions and the health of the processor. Such an approach, when handled correctly, yields a control structure that can degrade much more gracefully when a core fails. As nodes fail, the totality of the remaining computational resources can focus on keeping the higher-criticality tasks running, shedding the less vital functions as

This work was partially supported by the National Science Foundation, under grant CNS-1329831.
Author's addresses: Y. Xu, I. Koren and C. M. Krishna, Electrical and Computer Engineering Department, University of Massachusetts Amherst;

DOI: http://dx.doi.org/10.1145/0000000.0000000

necessary to make room. Recent years have seen the emergence of a theory of scheduling in such mixed-criticality platforms [Vestal 2007][Burns and Davis 2015]. Rather than a collection of dedicated microcontrollers, the trend is now towards a distributed, flexible, computation platform composed of multiple, often high-capability, processors.

For obvious reasons, *fault-tolerance* (FT) is needed for life-critical applications. Traditional fault tolerance that uses massive redundancy [Koren and Krishna 2007] can impose a considerable and unnecessary computational burden on the system. Designers have therefore turned to adaptive fault-tolerance [Krishna and Koren 2013][Krishna 2015][Liu et al. 2008] where the provided level of fault-tolerance is dynamically adapted to the current needs of the physical plant. These needs are a function of the current plant state and, consequently, vary with time and circumstance.

Adaptive fault-tolerance allows us to provide fault-tolerance on an as-needed basis and has the potential to reduce the size of the computational platform. It can also result in lower processor operating temperatures. Since processor failure rate increase exponentially with temperature, this often has a significant impact on its reliability.

In order to effectively implement adaptive fault-tolerance, the controlled plant dynamics have to be analyzed along with domain-specific knowledge of safety requirements to indicate the appropriate level of fault-tolerance at any given time. This paper describes a simulation framework, AdaFT, to accomplish this. AdaFT generates offline a table that allows the system, while in operation, to select the appropriate level of fault-tolerance. It does this by carrying out extensive offline analysis of the controlled plant dynamics and then uses machine learning techniques to express them as simple selection rules. Finally, the framework allows the designer to evaluate the impact on reliability of the thermal stress associated with the fault-tolerant workload. Overall, AdaFT allows us to increase the system's lifetime, and conversely, for a given desired life time, it reduces the amount of redundancy required.

This paper makes the following contributions. It introduces a tool, AdaFT, to identify the appropriate level of fault-tolerance as a function of the current plant state. Machine learning techniques are used for a classification process whereby this function can be compactly expressed. The tool allows for sensor noise as well as reduced-order models. It allows the user to study the fault-tolerance implications of control task dispatch frequency and processor response time.

This paper is organized as follows. In Section 2, the technical background is provided; this includes a state-space approach to system control and a discussion of thermally induced circuit aging. This sets the stage for a description of the software framework in Section 3. Section 4 describes the technical implementation of AdaFT. Section 5 illustrates the way to construct a simulated fault tolerant CPS using the AdaFT interface. Section 6 presents several case studies. Section 7 compares this work to prior work. Section 8 brings the paper to a close.

2. TECHNICAL BACKGROUND

2.1. Failures in Cyber-Physical Systems

It has long been understood that failures in CPS can be treated in a more application-specific way than failures in general-purpose systems. Meyer's *performability* specifies accomplishment levels for the controlled plant and calculates the probability that the controller will function well enough to meet these accomplishment levels [Goyal and Tantawi 1987][Meyer 1982][Meyer et al. 1980]. Another approach relies on the fact that the controller is in the feedback loop of the controlled plant [Shin et al. 1985][Krishna and Shin 1987]. Therefore, any computational delay contributes to the feedback delay. The impact of feedback delay on the plant performance is well understood in control theory. By quantifying this impact on the plant performance, one can obtain

cost functions to express the degradation of the quality of control. Obviously, the state of the controlled plant affects the impact of feedback delay on the quality of control.

Note that this approach presupposes the existence of sufficiently accurate models of the controlled plant. Such models would be needed, in any case, to assess the effectiveness of any control algorithm used.

2.2. A State-Space Approach to CPS Failure

A Cyber-side failure in a CPS happens when the controller is unable to keep the controlled plant within a designated subset of its state-space, called the *Safe State Space*, S^3. Full details can be found in [Krishna 2015]; here only the necessary foundation is provided to follow the rest of the paper. S^3 is defined as follows [Krishna and Koren 2013]: based on the application, the user or application engineer (or application-domain specialist) can specify the constraints that the plant must satisfy in order to be considered to be operating safely. These are called the *Safety Space Constraints* (SSC). For example, the maximum allowed G-force on an aircraft, together with the aircraft dynamics, can be used to specify constraints on the pitch, yaw and roll as well as the rate of change of these variables.

A point is in S^3 if (a) the plant satisfies the SSCs at the present time and, (b) based on the plant control laws, the control algorithm used, the actuator limitations, the control task execution policy and rates, and the specified limits of the operating environment impact on the plant, the plant will continue to satisfy these constraints up to a given horizon, as long as the correct control inputs are applied.

The impact of an erroneous controller output on the plant performance depends on the current plant state. If the plant is deep within its safe region of the state space, it may well be able to withstand a certain number of erroneous inputs without impairing safety. Such application error-tolerance translates to a lowered requirement for controller fault-tolerance.

2.3. Adaptive Fault-Tolerance

We will now show how the state-space approach leads naturally to adaptive fault-tolerance. We define three sub-spaces within S^3 as follows:

S_1**:** No fault-tolerance is required. The controlled plant is in a region of the state-space where even if the actuators are held at their worst-case incorrect setting until the next iteration of the control task, the plant will not leave its S^3. Hence, only one copy of the control task needs to be executed. Even if the task fails and produces the worst possible incorrect control output value, the plant remains safe and can be recovered in later periods.

S_2**:** It is sufficient for the controller to be fail-stop, i.e., the system generates only two types of controller output: correct or default (e.g., zero) output. Only error detection rather than error correction is needed in the control output calculation. For instance, one could use a processor duplex with two independent control calculations being compared. If a significant mismatch (i.e., outside the range of numerical approximations) is detected between the two outputs, then an error is declared in the computation and a zero control input (or other default value) can be applied.

S_3**:** Full fault-masking is required. If the controller produces an incorrect output, the plant cannot be guaranteed to stay in the safe state-space. Therefore full-strength fault- tolerance with fault-masking should be used, e.g., a triplex with majority voting [Koren and Krishna 2007].

Note that all other states outside of S_1, S_2 and S_3, i.e., outside of S^3, are either physically unachievable, or uncontrollable even by a perfect controller. The latter means that even if an always correct control input is applied, the physical plant might still

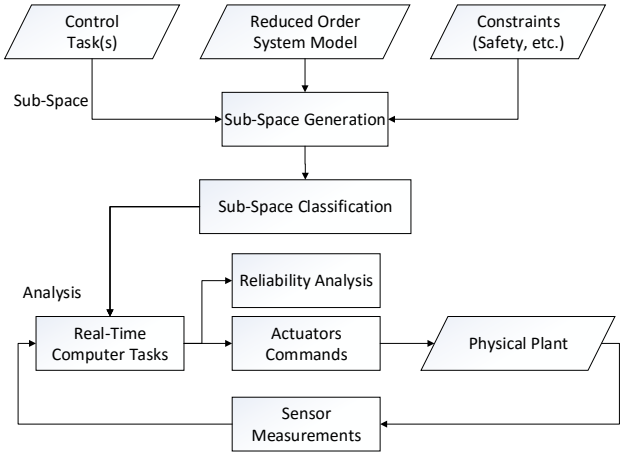


Fig. 1. Software Architecture of AdaFT

enter the unsafe region violating the SSC. In such a case, it still needs the full level of fault tolerance, but it is not guaranteed to be always safe.

With these sub-spaces, a *state-based adaptive fault tolerance* can be developed. Since the controlled plant is in S_1 for most of the time, a lower level of fault tolerance can be used, to reduce the amount of stress on the controller or to use the available released computational capacity for other tasks [Krishna 2015].

2.4. Impact on Thermal Age Acceleration

It is well known that the workload affects processor reliability. With a higher workload, the thermally-induced failure rate increases [Krishna 2015][Moazzami et al. 1989][Schoen 1980][Schroder 2007]. Operating at higher temperatures accelerates the device aging process. The rate at which such aging occurs can be captured by means of the *Thermal Age Acceleration Factor* (TAAF). If the TAAF over some time interval δt is α, the effective aging of the circuit over that interval is $\alpha\delta t$. α is a strongly increasing, non-linear function of temperature. The reliability advantage of adaptive fault-tolerance is that its lower computational burden reduces the average operating temperature and hence the amount of circuit aging.

3. STRUCTURE OF THE ADAFT FRAMEWORK

The overall structure of AdaFT is shown in Figure 1. It consists of two major parts: sub-space generation/classification and analysis. The first part focuses on the generation of the sub-spaces and the machine learning approach for sub-spaces classification. The second part takes the outputs from the sub-space classifier and simulates the system with reliability analysis. AdaFT takes the physical side information of the controlled plant as input, implements the adaptive fault tolerance approach to guarantee the same safety level as the traditional approach would do, while keeping the computing resource usage as efficient as possible, thereby improving the long term reliability of the computing platform.

3.1. Sub-Spaces Part

The sub-spaces part uses a *system model* of the controlled plant, which is a mathematical description of its behavior, typically given as a set of differential equations. In appropriate instances, AdaFT can use a *reduced order model*, which will simplify the system model to a reasonable level while still maintaining a sufficient accuracy. A reduced order model has the potential to significantly reduce the total computation time. We will discuss how to obtain such a model in later sections.

Control tasks are the real-time control tasks that control one or more of the physical state components. We focus here on periodic tasks; sporadic tasks with the period replaced by a minimum invocation between successive tasks can be incorporated easily. Typically, each task will have a period, deadline, *worst case execution time* (WCET), and power consumption. These parameters are usually determined during design time and hence are known in advance.

Constraints are the safety space constraints SSC, such as the minimum inter-vehicle spacing for *adaptive cruise control* (ACC) system, or the allowed angle range for the various joints of a robot.

Sub-space generation is one of the core parts of AdaFT dividing the operating space of the CPS into the S_1, S_2, and S_3 sub-spaces. These sub-spaces will in turn determine the level of run-time deployment of the fault tolerance (FT) needed to ensure the system safety.

Sub-Spaces classification takes as inputs points from the sub-spaces, and then uses machine learning techniques for their classification. The purpose of this part is to efficiently compute the FT level online, as the number of samples representing the sub-spaces is typically very large. Common machine learning algorithms only need limited memory space to store the fitted model to check, in real-time, which subspace a given point is in. The running time for these predictions is typically of the order of milliseconds, or even less [Murphy 2013].

3.2. Analysis Part

The analysis part of AdaFT uses a *real-time computer* model, with the support of a *real-time scheduling* policy along with adaptive fault tolerance to estimate the computational sub-system reliability.

4. IMPLEMENTATION

We have implemented AdaFT in both Python and Matlab. We chose Matlab due to its popularity in many engineering domains. Python is gaining popularity due to its powerful numerical and machine learning libraries such as *numpy*, *scipy* and *sklearn*. In this section we sketch several of the technically interesting aspects of the implementation.

4.1. Sub-Space Generation

The sub-space generator is the core component of AdaFT. It takes as inputs the control tasks, the system dynamics model, and the SSC safety constraints. These constraints are provided by the user and reflect application requirements. Then, for each given state in the state space, AdaFT determines whether it is in S^3 by simulating the system to a certain time horizon or final condition. The three sub-spaces, S_1, S_2 and S_3 are then generated from S^3.

Algorithm 1 shows how to generate S^3 through simulation based on the system dynamics. Each intermediate state before the time horizon (or final condition) is checked against SSC. If all are within SSC, then this specific initial state is within S^3.

ALGORITHM 1: S^3 Generation

Input: The operational state space
Output: S^3;
for *All x that satisfy the SSC* **do**
    run simulation with x as initial condition, with control period (step length) δt and correct control, until the time horizon is reached or the final condition is satisfied;
    if *all intermediate states satisfy the SSC* **then**
        S^3.add(x);
    end
end

ALGORITHM 2: Sub-spaces Generation

Input: S^3;
Output: S_1, S_2, S_3;
for *All x in S^3* **do**
    run simulation with x as initial state with control period δt with the worst case wrong control;
    if *after one period the state is within S^3* **then**
        S_1.add(x);
    else
        run the same simulation with zero control;
        if *after one period the state is within S^3* **then**
            S_2.add(x);
        else
            S_3.add(x);
        end
    end
end

Algorithm 2 shows how to generate S_1, S_2 and S_3. It takes S^3 as input and for each state x in S^3, it simulates the controlled plant for one task period. For S_1, the worst case wrong control is applied, whereas to get S_2 a zero control representing a fail-stop controller is applied. If after one task period, the physical state of the plant is still within S^3 even with the worst case wrong control input, this state x is within S_1; otherwise if the plant is in S^3 with zero control, x belongs to S_2. Finally, x is in S_3 if it is in neither S_1 nor S_2.

Remark 1: It should be noted that both hardware and software fault tolerance can be applied using AdaFT. We have already discussed hardware fault tolerance using duplex or triplex. Software fault tolerance techniques are similar as they also rely on the use of redundancy. For example, *N-version programming* is a forward error masking technique. Another example of an error detection and recovery technique is the *recovery block approach* [Koren and Krishna 2007]. All AdaFT needs to know is what sub-space the physical plant is currently in. If the plant is in S_1, no FT is needed (neither hardware or software); for S_2, duplex for hardware FT and/or error detection, e.g., a 2-version software, is sufficient, followed by a default control input for a fail stop model; finally, for S_3, a triplex for hardware FT and/or a 3-version programming for software FT may be used. The user can decide whether to use hardware only, or hardware and software fault tolerance together.

Remark 2: The complexity of this approach is proportional to the number of voxels of the state space that are evaluated. This number is obviously exponential in the number

of state space dimensions. However, we do not require that the entire safe state space be evaluated. Each voxel in S^3 starts, by default, in S_3; it may be reclassified as in S_1 or S_2 following an evaluation. For this approach to be useful, it is sufficient to evaluate the more frequently visited parts of the state space, which can be obtained by gathering traces of the state space trajectory and evaluating the state space neighborhoods of these points.

Remark 3: Note that we do not explicitly model communication faults. Highly effective coding and other redundancy mechanisms exist to reduce communication errors to desired levels. If necessary, the event of an undetected/uncorrected error in communication can be included in the failure probability of the relevant task.

4.2. Worst Case Controller/Actuator Action

The user has to specify a cost function indicating divergence from the desired state trajectory or value. Control is usually applied to ensure minimization of such a function. However, one can instead try to maximize such a function given the control input constraints. If such a maximum divergence still keeps the controlled plant within S^3, that point will be declared to be within S_1. A good rule of thumb for this type of cost function is to use $cost = \frac{1}{n} \times \sum_i (x_i - x_{id})^2$, where x_i is the actual value for the ith state which we care about its safety, and x_{id} is the corresponding desired value. Note that scaling of the state variables is important, since the total cost will otherwise mainly consist of the states with large absolute values.

This approach to compute the worst case control is essentially a simplified optimization problem. The only constraints for this problem are the control input bounds, normally provided by the specifications of the actuators. Algorithms to solve optimization problems with potentially complicated state dynamics might become computationally expensive, however, since this step is executed offline using simulation software, execution time will not be an issue.

It should be noted that for many applications it is sufficient to use some heuristics to determine the worst case control/actuator outputs. For example, in the ABS braking system, the controller or actuator will control the *slip ratio* towards its optimal point (a typical value is 0.2). If the current slip ratio is greater than this point, the worst case actuator command can be set to the upper bound of the actuator value with the opposite control direction.

4.3. Reduced Order System Model

For simpler controlled plants it is feasible to use a full order system model to generate the sub-spaces. For more complex plants, we use machine learning techniques, for example, *feature selection*, and *precision-recall* trade-off [Murphy 2013]. We first sample certain amount of data with a coarse granularity, run simulations, and classify these samples as to whether or not they violate the SSC. We split the resulting data set into training and testing sets. The training set is for fitting a particular machine learning model, while the testing set is for testing the prediction performance of unseen data using the fitted model [Murphy 2013]. The next step is to use a machine learning algorithm that can calculate he importance of each feature, e.g. *random forest*, to fit the data. At this point, if the testing accuracy is higher than a threshold, (e.g., higher than 98%), and the *precision*, i.e., the percentage of the correct prediction of the class belonging to S^3, is also high enough, (e.g., higher than 99%), we can assume the fitted machine learning model can not only fit the training data very well, but can also be well generalized. We can then use this simplified fitted model, rather than the real simulations which are more time consuming and complex, to generate more data for S^3.

On the other hand, if we observe a high training accuracy with a lower testing accuracy, this typically means an overfitting of the model, or *high variance*. We must either reduce the model complexity or get more data. For the first approach, we can remove the features with relatively low importance from the model. For the latter, we can sample more data with respect to the more important features, again according to the feature importance. If there is a low training accuracy, which means an underfitting or *high bias*, we must first fit the training set using more advanced techniques such as the *ensemble* model that combines several weaker models to achieve a better one. Our results show that most CPSs have a high training accuracy due to the clear relationship between the inputs (the previous state values and the control inputs) and the outputs (the new state values), determined by differential equations. In all of our case studies, we achieved very high training and testing performance. If, for any case, it is impossible to achieve high accuracy, we need to sacrifice in terms of *recall* which is the proportion of true positives (data points belonging to S^3) that are correctly predicted as such, for high *precision*. With a slightly lower recall, the system might waste some computing resources for providing unnecessary redundancy; but with a low precision, a hazardous behavior may occur.

It should be noted that other approaches can be used along with the machine learning. For example, since each simulation is independent of others, parallel computing can be employed to accelerate the data collecting process using multiple computers. Further, pruning techniques like *Branch-and-Bound* can be used [Clausen 1999].

We now consider the inverted pendulum [Wittenmark 2011] as an example to demonstrate the machine learning approach. The system consists of an inverted pendulum mounted on a motorized cart and the pendulum is kept close to vertical by controlling the cart speed. The system has four states: (cart position, cart velocity, pendulum angle, and pendulum angular rate). We first generate data with the following granularity: the step size of the cart position, cart velocity, and pendulum angular rate are all set to 2 while the step size of the pendulum angle is 0.1. We define a reasonable operating range for each state component, e.g., (-10, 10), (-10, 10), (-0.5, 0.5), (-10, 10), respectively. For example, we generate data points for pendulum angles from -0.5 to 0.5 radians, with a step size of 0.1 radians. There would be around 10000 data points, which are then simulated, and labeled as 1 if a particular data point never violates the safety constraint (the angle stays in the range of -0.5 to 0.5 radians), and 0 otherwise.

We use 20% of the data as the testing set, and use the *random forest* technique with *k-fold cross validation* to fit the remaining 80% of the data. During the process, we use a grid search to tune the hyper-parameters of the random forest, i.e., the number of trees/estimators.

For the initial data set we have achieved a 99% accuracy for the training set and a 95% for the testing set. The training accuracy is higher than the testing accuracy indicating overfitting. We need to either reduce the complexity or get more data. The feature importance for the four state variables was: 0.285, 0.301, 0.155, and 0.258, from which we can conclude that all of the four variables are significant for the classification. Therefore, we followed the second approach and get more data. We decreased the granularity of the cart velocity from 2 to 1, since it is the most important feature for this particular problem. With the new 20000 data points, we obtained a training accuracy of 100%, a testing accuracy of 99.7% and a testing precision of 99.5%. The random forest technique achieved a good training and testing accuracy, as well as a high precision so we then used it to generate data for S^3.

4.4. Actuator Noise, Sensor Noise and Failures

Controlled plants are subject to noise or uncertainties from the operating environment. For actuator noise, AdaFT considers the worst case scenario when generating the sub-

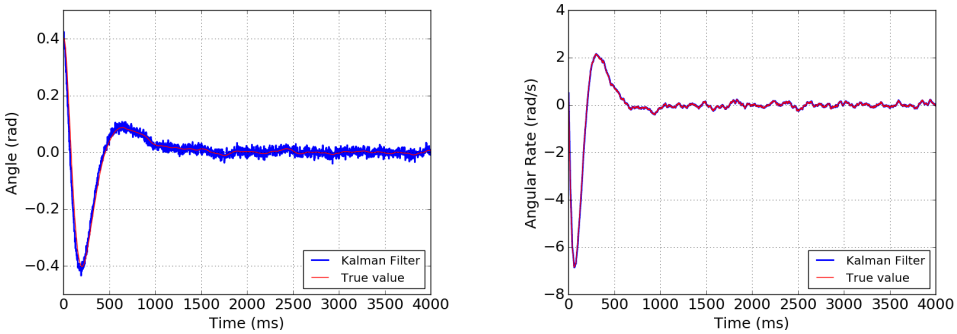
spaces. In AdaFT, the motion model of the controlled plant includes a *state transition probability* $p(x_t|u_t, x_{t-1})$, where x_t, x_{t-1} are the state values at time t and $t-1$, respectively, and u_t is the control input at time t. These probability distributions are derived using the input models of the noise. For a particular final condition provided by the simulation, AdaFT checks, up to a specified confidence interval, whether all states are safe, and if so, the initial state is declared to belong to the corresponding sub-space.

As for the sensor noise, there are many well studied techniques for noise filtering, among which the *Kalman filter* and the *Particle filter* are commonly used in control applications, such as self-driving cars and UAVs [Thrun et al. 2005]. Both techniques are *dynamic Bayesian networks* (DBN). The Kalman filter is an exact tracking algorithm, while the Particle filter is an approximate one. Both first use the system dynamics and the control inputs to generate a prior belief about the physical states. This is called the *prediction step*. Then, they calculate the likelihood of the sensor measurements given the initial prior belief. Finally, a posterior belief distribution is obtained for the updated estimation. This is called the *update step*. The Kalman filter produces optimal estimates for unimodal linear systems with Gaussian noise. It calculates a *Kalman gain* which will be used during the update step.

In contrast, the Particle filter uses *Monte Carlo* sampling to randomly generate particles, each corresponding to an initial guess. Then, during the prediction step, it moves the particles based on the dynamics model to obtain the next state of each particle. At the update step, the Particle filter updates the *weight* of each particle based on the sensor readings, which is essentially the likelihood of the sensor reading for each particle. Particles that closely match the readings are weighted higher than those which do not match well. Finally, the Particle filter uses a *resampling* technique to discard highly improbable particles and replaces them with copies of the more probable ones, in order to get the posterior belief distributions. The Particle filter works well for nonlinear systems, whereas the Kalman filter must first perform linearization which might be difficult for some systems. The detailed mathematical derivations of these algorithms can be found in [Thrun et al. 2005].

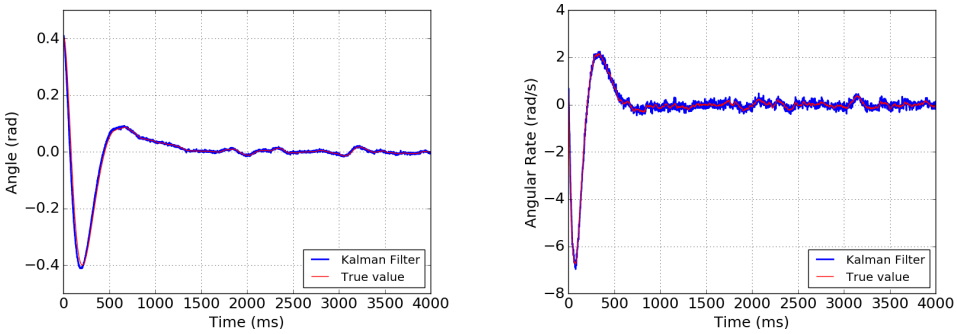
AdaFT uses the Kalman filter to track linear systems, and the Particle filter for non-linear systems. We used the Kalman filter for the inverted pendulum mentioned earlier. The initial state conditions are set to (0, 0, 0.4, 0.5) and the actuator noise standard deviation to 0.06 N. We used two sensors with different profiles for the tracking and assume that both sensors have the ability to measure angle and angular rate. The two sensors have an angle noise standard deviation of 0.01 rad and 0.002 rad, respectively, and have an angular rate noise standard deviation of 0.005 rad/s and 0.1 rad/s, respectively. We assume that sensor 1 is better at sensing angular rate, while sensor 2 is better at sensing angles. Figures (2 - 4) show how filtering algorithms can reduce the sensor noise, and improve the tracking accuracy and confidence.

In order to handle transient and persistent sensor failures, AdaFT extends the standard Kalman and Particle filter algorithms. As discussed before, during the update step, these filtering algorithms calculate the likelihood of the sensor measurements given the prior beliefs. For Particle filters, the calculation of *weight* for each particle is the likelihood calculation. It is easy to calculate the likelihood of the sensor measurements given the output of its prediction step as the prior belief. If the calculated likelihood is less than a reasonable threshold (e.g., less than 1%), it is highly likely that the sensor has given a wrong value. In such a case, the system will skip the update step and take the value from the prediction step for this particular sensor. Intuitively, this approach assumes that the belief of the system has a certain amount of inertia that would overcome the temporary sensor failure. With regard to a persistent sensor failure, this approach would consistently use the prior belief or the remaining working sensor(s) for the tracking. Figure 5 shows how the filtering algorithms would handle



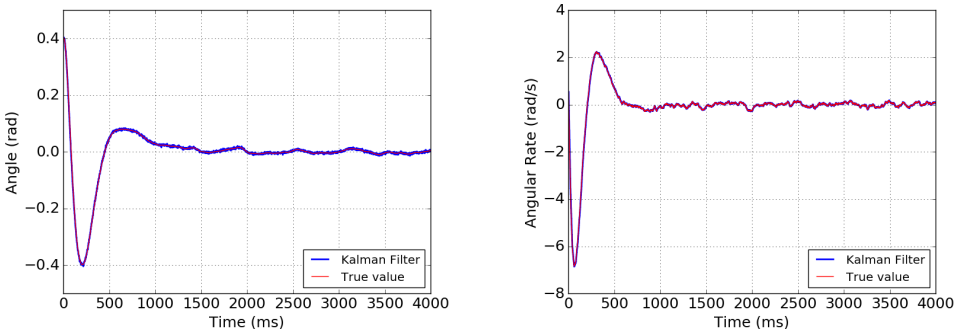
(a) Angle Tracking (b) Angular Rate Tracking

Fig. 2. Kalman Filter for Sensor 1 ((a)-(b))



(a) Angle Tracking (b) Angular Rate Tracking

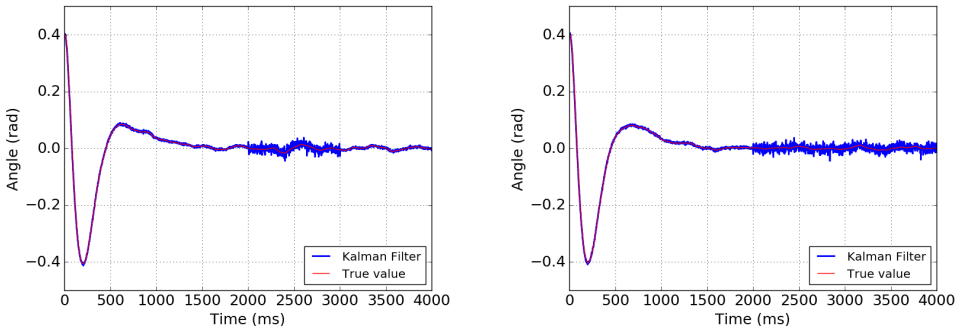
Fig. 3. Kalman Filter for Sensor 2 ((a)-(b))



(a) Angle Tracking (b) Angular Rate Tracking

Fig. 4. Kalman Filter for the two sensors combined ((a)-(b))

sensor failures. We deliberately assigned some arbitrary wrong value (100) to sensor 2 for 1 second for the transient failure case, and for the remaining of the simulation for



(a) Transient Failure (b) Persistent Failure

Fig. 5. Kalman Filter with a Sensor 2 Failure ((a)-(b))

the persistent failure case. We see from the figures that the recovery from transient failures can be very fast, and the performance degradation is not very severe.

4.5. Real-Time Computing Model

AdaFT has a built-in real-time task model with the following attributes: name, period, deadline, *worst case execution time* (WCET), *actual execution time*, power, and status. It also has a probability density function of the execution time, in order to randomly generate the actual execution time for simulation purposes. The period and deadline of the task is for real-time scheduling, and the power is for thermal aging (TAAF) analysis.

Real-time scheduling is an essential part of managing a real-time system. There are two widely used scheduling algorithms, i.e. *Rate Monotonic* (RM) and *Earliest Deadline First* (EDF). RM assigns periodic task priorities as inversely proportional to the task periods [Lehoczky et al. 1989][Cooling 2013]. By contrast, EDF determines the task priorities according to their absolute deadlines.

AdaFT has scheduling modules to support both RM and EDF. When users develop their CPS using AdaFT, they must guarantee the system schedulability, i.e., meet all task deadlines. RM and EDF have have been studied for schedulability and interested readers can refer to [Cooling 2013] for the details, as well as a comprehensive survey of other real-time scheduling algorithms.

With a given real-time scheduling algorithm the user can easily experiment with different periods and possibly different execution times of each control task, and see the impact of these parameters on the size of each sub-space.

4.6. Estimating Thermally-Induced Aging

TAAF expresses by how much the natural circuit aging process is accelerated by operating at a high temperature [Krishna 2015]. AdaFT uses a first-order thermal model to estimate temperatures; if desired, this model can be replaced by the user with one that more precisely captures the thermal characteristics and the failure dependencies of the particular hardware.

Each processor in AdaFT is treated as a single node, dissipating $p(t)$ Watts at time t. A standard equivalent electrical circuit model is used to model heat flow, where resistances and capacitances have thermal counterparts [Skadron et al. 2004]. Thermal capacitance is the amount of heat required to raise the temperature of a node by one degree; thermal resistance determines the heat flow across a given temperature gradi-

ent (temperature is treated as an analogue of voltage). Denote by R and C the thermal resistance (associated with heat flow from the node to the ambient) and capacitance (of the node), respectively. Let $T_{proc}(t)$ and T_{amb} denote the absolute temperature of the processor and the ambient temperature, respectively. Then, the following differential equation emerges from the equivalent circuit model: $C\frac{dT_{proc}(t)}{dt} = p(t) - \frac{T_{proc}(t)-T_{amb}}{R}$. Solving this yields the temperature at any given time as a function of the power consumption.

The aging acceleration model for the hardware depends on the actual technology used. AdaFT provides the option to define a software module which expresses this function; however, a default aging module is provided based on the widely used Arrhenius acceleration model, in which the aging factor at time t, $\lambda(t)$, is proportional to $\exp(-E_a/(kT_{proc}(t)))$ [Escobar and Meeker 2006]. Here, E_a is the *activation energy* [Vigrass 2004], whose value is a user-provided input. The accumulated aging over a given interval $[a, b]$ is then calculated as $\int_a^b \lambda(t)dt$.

AdaFT computes TAAF based on the power consumed as a function of the load. The hardware configuration consists of three or more cores/processors on which tasks can be scheduled. The default scheduling policy is to pick the coolest processor to run at each time step; but the user can replace this scheduling algorithm by any other. AdaFT then computes the average TAAF as well as the instantaneous TAAF for each core. Recall that when the controlled plant is in sub-space S_i, it schedules i copies or versions of the control task.

It should be noted that TAAF is closely related to a more common term in the fault tolerance and reliability literature, i.e., the *mean time to failure* (MTTF), which is calculated as: $\int_0^\infty tf(t)dt$, where the $f(t)$ is the probability distribution density of the lifetime under unstressed conditions. Therefore, if the effective age of the device at chronological time t is given by $x(t) = \int_0^t \lambda(\tau)d\tau$, the updated MTTF is given by: $\int_0^\infty tf(x(t))dt$. Once TAAF is calculated, AdaFT uses these equations to compute the MTTF.

Remark 4: Heating is by no means the only accelerator of failure. Other stressors include humidity, mechanical vibration and static discharge. Our focus in AdaFT is on allocating and scheduling computational workload which primarily affects device temperature. Other stressors have to be dealt with by other, orthogonal, means, such as improved packaging, mechanical damping and changes in circuitry; their impact on reliability can be modeled separately.

4.7. Sub-space Classification

During operation, the application must rapidly determine which sub-spaces it is in. Since such a real-time classification can never guarantee a 100% accuracy, a conservative approach should be developed for system safety. The system might be allowed to make a few wrong decisions from S_1 to S_2 or even to S_3, but not the other way around. Mis-classification from S_1 to a higher level of fault tolerance will do no harm to the system safety, only waste some resources.

Since the plant state-space is well defined, we can treat this as a supervised classification problem [Murphy 2013]. We first perform some pre-processing such as feature scaling, feature selection or extraction using *principal component analysis* (PCA) [Murphy 2013]. We then employ a machine learning classification scheme, for example, *random forest* (RF), *logistic regression* (LR), *neural network* (NN) or *support vector machine* (SVM) with various kernel functions, including linear, polynomial and Gaussian kernels [Murphy 2013]. Each of these algorithms has several hyper-parameters to tune, such as the number of trees for random forest or the regularization strength for LR, NN and SVM. We use the technique of *grid search* to find the most appropriate algorithm with the best combination of hyper-parameters. Sometimes it is necessary

to use an ensemble approach to find the best machine learning scheme, i.e., use several individual schemes combined with a majority voting. It should be noted that the most suitable machine learning algorithm is application specific. The purpose of the AdaFT interface is to select the best machine learning scheme for the the particular application, to be used in the analysis part of AdaFT. We refer the reader to [Murphy 2013] for a detailed explanation on how these algorithms work.

Dealing with safety critical issues: As discussed before, we must guarantee high *precision* but may sacrifice some of the *recall*. One approach is to adjust the threshold value used to make decisions. Normally, the learning algorithm will produce a 1, for a two-class classification problem, if the output of the hypothesis function is larger than a threshold of 0.5, and a 0 otherwise. In multi-class classification problems, the algorithm will pick the class with the largest output of the hypothesis function. These probabilistic values show how confidently the algorithm makes certain decisions. If the confidence level of the algorithm needs to be increased, this threshold can be adjusted from 0.5 to a higher value. If there is any wrong classification from a more dangerous sub-space (e.g., S_3) to a safer sub-space (e.g., S_2 or S_1), the threshold value should be adjusted. If the largest value among all classes from the hypothesis function is higher than the threshold value, the algorithm will take that value and make the corresponding decisions; otherwise, it will determine the current system state to belong to S_3. Some machine learning libraries such as *scikit learn* provide automatic methods that can be used to find the best threshold value, such as the precision-recall curve [Buitinck et al. 2013].

5. ADAFT PROGRAMMING INTERFACE

Figure 6 shows the major parts of AdaFT, in a UML class diagram. AdaFT provides abstractions such as the sub-generator, processed through a language-integrated API in both Python and Matlab. For example, the *CPS* class wraps all major components of a CPS, including the *Physical System*, *Sensor*, *Cyber System*, and *Actuator*. The *Sub-Space Generator* class uses the *CPS* to generate all of the sub-spaces that are then used by the *Classifier* to fit a classification model, which is later used by the *CPS* class for execution. Before the sub-spaces and the fitted model are generated, *CPS* runs to generate the sub-spaces; once the fitted model as a *classifier* is present in *CPS*, it runs with the classifier for reliability analysis. The *Cyber System* includes one or more *Processor* objects, with support from the Real-Time Operating System *RTOS*, which in turn consists of several *Task Model* objects as real-time tasks. The *Worst Control* is a child-class of the Task Model. The *Reliability Model* and its child-class *TAAF* are also part of the *Processor* class.

To use AdaFT, the user should write a physical system implementation, possibly inherited from the *PhysicalSystem* class. In particular, the *update()* and *issafe()* methods need to be implemented. *update()* evolves the physical state vector, according to control inputs and the corresponding actuator signals. *issafe()* checks if the SSC is satisfied during simulation.

The next step is to implement the control tasks, for both correct and worst case controls, through the API from the *TaskModel* class. Essential attributes of a task need to be specified, including: name, power, WCET, deadline, and period. In addition, the *run()* abstract method must be implemented, which is the actual algorithm of the task. Note that the filtering algorithms discussed before are also real-time tasks and should be run with the highest available redundancy, since the physical side information will be estimated through them. The inputs of the custom control tasks should be the outputs of these filtering tasks, whose inputs are the raw sensor readings.

There are additional methods that users can implement or override, such as the heuristics to sort the data points for the sub-space generation, according to some safety

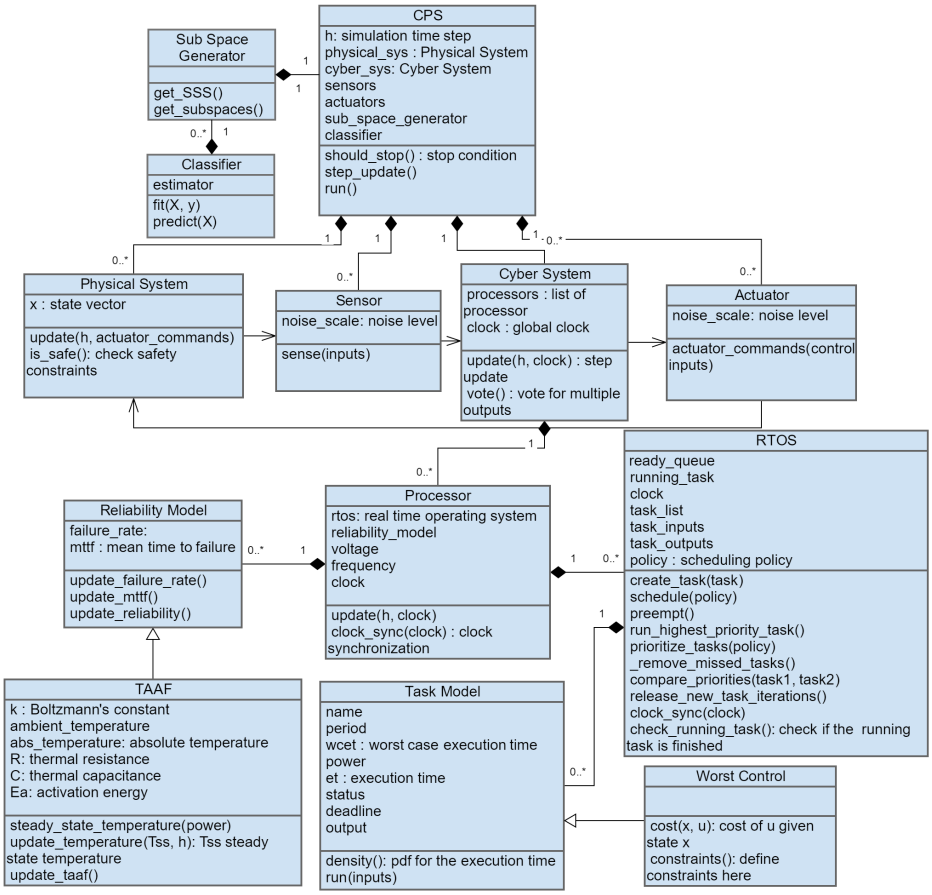


Fig. 6. Class Diagram of AdaFT

rules, but they are not required, either because they are not the core parts of AdaFT, or AdaFT already has default implementations.

If the system dimension is small, AdaFT will start the whole process to generate the sub-space through *getSSS()* and *getSubspaces()* methods. Otherwise, the user can provide a fitted machine learning model, discussed in Section 4.3, as the input to the *getSSS()* method to generate S^3.

After the sub-spaces S_1, S_2 and S_3 are generated, the user must fit a machine learning model for the classification through the API from the *Classifier* class. This fitted model will then be used by the analysis part of AdaFT for reliability analysis. Other additional custom parameters that the user can specify include, for example, sensor and actuator noise, processor voltage and frequency, and a real-time scheduling policy for RTOS.

5.1. Example Program

The inverted pendulum example is a linear system with a Kalman filter, and *A, B, C, D* matrices to define and track the trajectory of the physical states. Detailed equations can be found in the case study section. Here we demonstrate how to program such an adaptive fault tolerant system using AdaFT.

To provide the system dynamics and the control task (LQR), we use two child classes inherited from their corresponding parent classes. Note that for the worst case control, if the worst case output can be determined without solving an optimization problem, the user can override the *run()* method to directly provide this output, since the *WorstControl* class itself is a child class of *TaskModel*.

```
class Pendulum(adaft.PhysicalSystem):
  def update(self, h, clock, u):
    self.x = # update the state vector according to actuator input u.

  def is_safe(self):
    return -0.5 <= self.x[2] <= 0.5
        # outside this range is unsafe

class LQRInvPen(adaft.TaskModel):
  def run(self, inputs):
    # inputs are the outputs from the Kalman filter
    # this is the control input from LQR algorithm
    self.output = -np.dot(self.K, inputs)

class WorstLQRInvPen(adaft.WorstControl):
  def cost(self, x, u):
    new_x = # get new state value given control u and current state x
    return 1/2 * ((new_x[2] - desired_x) ** 2)

  def constraints(self):
    self.constraints = {# define constraints here}
```

Users can specify sensor and actuator noise and put these objects into the *CPS* driver class, to start generating sub-spaces.

```
sensor_noise = # specify sensor noise
actuator_noise = # actuator noise

sensor = adaft.Sensor(sensor_noise)
actuator = adaft.Actuator(actuator_noise)

lqr = LQRInvPen()
worst_lqr = WorstLQRInvPen()
task_list = {lqr.name:lqr}
rtos = adaft.RTOS(task_list)
# We assume 3 processors for redundancy
processors = [adaft.Processor(rtos) for i in range(3)]
cyber = adaft.CyberSystem(processors)

cps = adaft.CPS(sensor, actuator, pendulum, cyber)
subspace = adaft.SubSpaceGenerator(cps, {lqr.name:worst_lqr})
subspace.get_SSS()
subspace.get_subspaces()

clf = adaft.Classifier()
# use machine learning techniques here
clf.fit(subspace.subspaces)
cps.classifer = clf

cps.stop_condition = # determine when the simulation should stop
cps.pendulum.x = # define initial state conditions here
cps.run()
```

```
# now we can calculate the TAAF and/or MTTF of each processor
plot(cps.cyber.processors[0].taaf)
print(cps.cyber.processors[0].mttf)
```

6. CASE STUDIES

In this section we present three case studies using AdaFT. The first one is a linear system, the second and third are non-linear with the third being a multi-agent system. We will use a data analytic approach to analyze Case Study 1. We will show the impact of different environmental conditions on sub-spaces, for Case Study 2. Finally, we will show how interactions between multiple agents can be studied with AdaFT.

6.1. Case Study 1: Computer Controlled Inverted Pendulum

The system in this example consists of an inverted pendulum mounted on a motorized cart. The inverted pendulum is a commonly used example in the control and CPS literature [Wittenmark 2011] since it is well understood and can be modeled into either a *linear time invariant* (LTI) or a nonlinear form. The LTI system can be represented using the following set of linear equations:

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned}$$

where $x(t), y(t)$ and $u(t)$ are the plant state, output vector, and input vector, respectively, and A, B, C, and D are the matrices defining the controlled plant.

The objective of the control system is to balance the inverted pendulum by applying a force to the cart that the pendulum is attached to. This case study uses an optimal control algorithm *linear quadratic regulator* (LQR) [Wittenmark 2011] to additionally control the cart's position to 0. In this case study, we make the following assumptions:

(1) The standard deviation of the actuator noise for both cart position and pendulum angle is 3 N.
(2) Sensor 1's noise standard deviation for the states of (pendulum angle, angular rate) is: (0.01 rad, 0.005 rad/s)
(3) Sensor 2's noise standard deviation for the states is: (0.002 rad, 0.1 rad/s)
(4) Initial condition for TAAF analysis is: (0 m, 0 m/s, 0.4 rad, 0.5 rad/s)
(5) Kalman filter is used for filtering out sensor noises.
(6) LQR control task period: 20 ms; deadline 20 ms; WCET: 1 ms
(7) Kalman filter task period: 2 ms; deadline 2 ms; WCET: 1 ms

State Equations: We refer the reader to [Michigan 2012] for the detailed mathematical state space equations for the system. Below is the summary of the linear equations:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\phi} \\ \ddot{\phi} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -0.1818 & 2.6727 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -0.4545 & 31.1818 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \phi \\ \dot{\phi} \end{bmatrix} + \begin{bmatrix} 0 \\ 1.8182 \\ 0 \\ 4.5455 \end{bmatrix} \vec{u} \tag{1}$$

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \phi \\ \dot{\phi} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \vec{u} \tag{2}$$

where x is the position of the cart and ϕ is the angle of the pendulum. Below is the K matrix that is used to compute the control input $\vec{u} = -K\vec{x}$, where $\vec{x}$ is the state vector:

$$K = [-61.993 \ \ -33.504 \ \ 95.06 \ \ 18.83] \tag{3}$$

Control Algorithm: We use the *lqr* function from Matlab to obtain the K matrix, which implements the LQR optimal control [Wittenmark 2011]. This function computes the control gain matrix K based on the system dynamics matrices, Q and R. The Q matrix defines the cost experienced when the states deviate from the desired ones, whereas the R matrix defines the cost when the control inputs are large [Wittenmark 2011].

Safety Space Constraints: We define the SSC to be: $-0.5 < \phi < 0.5$, where ϕ is the angle of the pendulum, as we do not want the pendulum's angle to be too large, otherwise it is unsafe. Although the system is also controlling the cart position, we do not consider this state as a safety critical state variable, but rather an additional factor affecting the quality of control.

To determine the cost function for the wrong controller or actuator output, we must first identify the desired states for the system. The objective of the inverted pendulum system is to keep the pendulum as close to vertical as possible. Therefore, the desired state of the pendulum angle is 0. Thus, we select the following cost function: $cost = (x_3 - x_{3d})^2$. This system has 4 state variables, but we only care about the third variable which is the pendulum angle.

Sub-Spaces and Classification Results: We followed the approach presented in Section 4.3 to generate S^3, which was then used to generate all three sub-spaces. Since the system has four dimensions, typical data analysis will include a *scatter plot matrix* to plot the pairwise relationship between each two variables. Due to space considerations we do not include these plots in the paper.

We have collected data from the following operating space of the system:

(1) Cart position: from -6 to 6 meters
(2) Cart velocity: from -6 to 6 m/s
(3) Pendulum angle: from -0.5 to 0.5 radians, outside which we considered as unsafe.
(4) Pendulum angular rate: from -6 to 6 rad/s

After obtaining the sub-spaces, we experimented with two cases regarding S_3. The first one is to only consider S_3 inside S^3, while the second one is to allow S_3 to include data points inside S^3 that do not belong to either S_1 or S_2, as well as all data points outside of S^3. If, for any reason, the state falls out of S^3, full level of fault tolerance will be required, which would rarely exist in practice as long as the control system is properly designed.

The results show that S^3 includes cart positions with the range from -1 to 1 meter, cart velocities from -4 to 4 m/s, pendulum angle -0.5 to 0.5 radians, and angular rate from -5 to 5 rad/s. Inside S^3, S_1 has pendulum angles from -0.2 to 0.2 radians, with which different combinations of the other three state vaiables exist from the same range of S^3. In other words, as long as the pendulum angle falls into the $[-0.2, 0.2]$ range, all points in S^3 for the other three state variables belong to S_1 with appropriate combinations. For example, if the pendulum angle is -0.2 radians, then a combination of cart position of 0, cart velocity of 0, and any point from -1 to 2 rad/s for the angular rate belong to S_1. Similarly, S_2 corresponds to a pendulum angle range from -0.35 to -0.2, as well as from 0.2 to 0.35.

For the sub-space classification, we use some common machine learning techniques to find the fitted model with the best performance. We focus on the accuracy here as the storage and prediction time do not vary much between different machine learning schemes. From our results, except simple logistic regression that has only 85% testing

	Feature Importance				Accuracy	
	Cart Position	Cart Velocity	Angle	Rate	Training	Testing
Entire Space	0.24	0.3	0.21	0.25	100%	99.7%
Within S^3	0.0038	0.01	0.97	0.012	100%	100%

Table I. Random Forest Performance for Inverted Pendulum

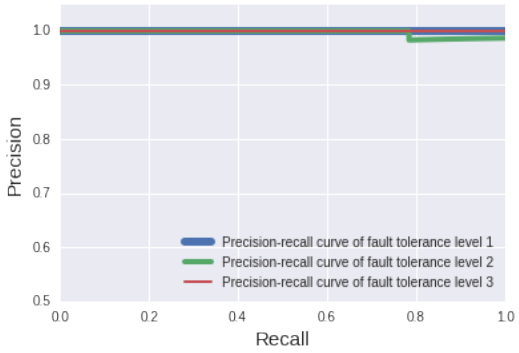


Fig. 7. Precision-Recall Curve for S^3

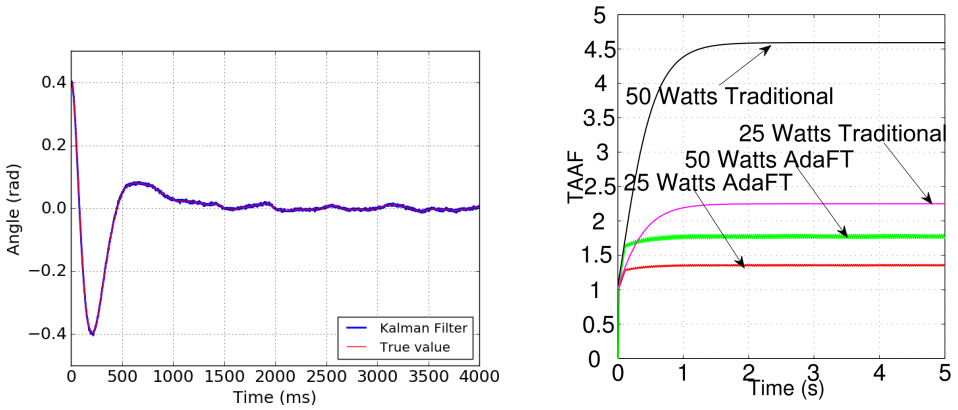
accuracy, most of other schemes have a score close to 100%. We only show here the performance of random forest, since it also allows us to calculate the feature importance. Table I shows that the fitted model using random forest has high testing accuracy for both cases, one with S_3 only inside S^3, and the other with S_3 in the entire space. For the latter, all four state variables play important roles, while for the S^3 only, the angle has the most significant impact on the sub-spaces.

Figure 7 shows the precision-recall curve for S^3. As expected, the precision is very high except when the recall for S_2 is higher than 80%, reducing it to 95%. AdaFT can adjust the prediction threshold to improve the precision. If we set the precision threshold to 99.5% for S_2, we have a total testing accuracy of 98.61%. These are the mis-classification from S_2 to S_3. The precision-recall curve for the entire space is similar and is omitted.

Reliability Results: We experimented with two different power consumptions for the control task: 25 and 50 watts. Figure 8(b) clearly shows that using AdaFT, there is a significant improvement in TAAF compared to traditional (non-adaptive) FT. Under our scheduling algorithm (picking the coolest processor to run the task), all three processors have similar values of TAAFs. Generally, the greater the power consumption, the more the beneficial impact of our adaptive approach.

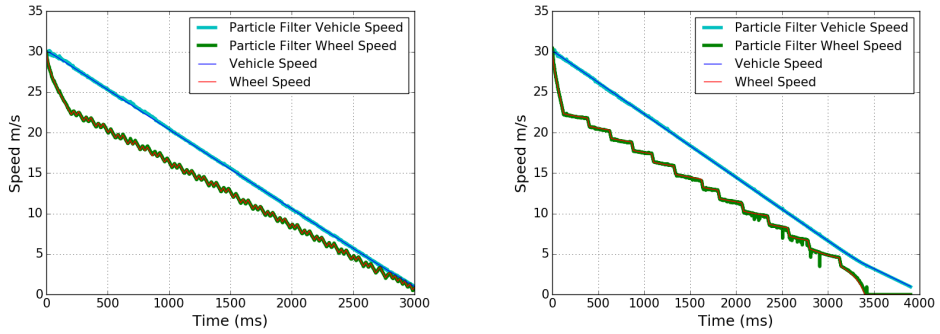
6.2. Case Study 2: Anti-lock Braking System (ABS) in a Straight Trajectory

For the case-study of an ABS in a straight line the canonical seven degrees-of-freedom car model was used [Rajamani 2011][Currier 2011], along with the Dugoff tire model [Dugoff et al. 1969]. The user inputs to AdaFT are the physical plant dynamics and the SSC. The car dynamics are nonlinear; the reader should see [Rajamani 2011][Currier 2011][Dugoff et al. 1969] for the detailed explanations of the car state equations and control algorithms. These equations allow us to implement the system using AdaFT API, similar to the template shown in Section 5.1. Below is the summary of our assumptions.



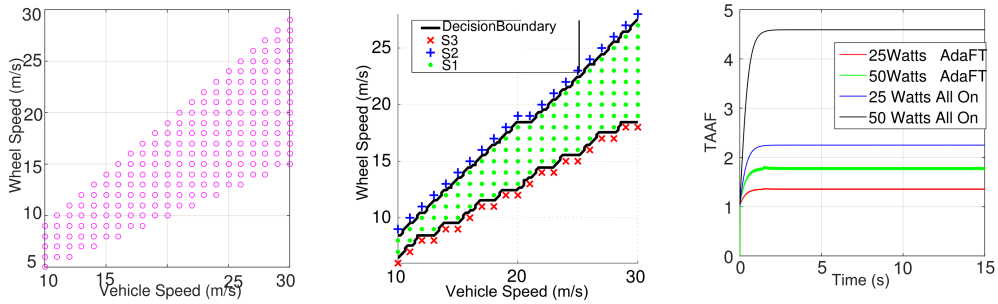
(a) Pendulum Angle (rad) (b) TAAF

Fig. 8. Inverted Pendulum's (a) angle and (b) TAAF



(a) Road Friction Coefficient of 1.0 (b) Road Friction Coefficient of 0.8

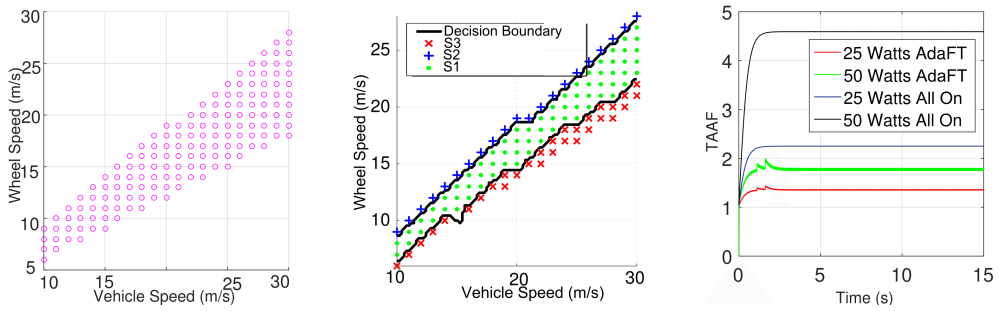
Fig. 9. ABS in a Straight Line for two road conditions



(a) SSC (b) FT Sub-Spaces (c) TAAF

Fig. 10. Sub-Spaces and TAAF of ABS in a Straight Line with a Road Friction Coefficient of 1.0

(1) Standard deviation of the actuator noise 5 N.



(a) SSC (b) FT Sub-Spaces (c) TAAF

Fig. 11. Sub Spaces and TAAF of ABS in a Straight Line with a Road Friction Coefficient of 0.8

(2) Sensor noise levels (standard deviation) for the two state variables (vehicle speed, wheel speed) are: $(0.5m/s, 0.5m/s)$. Note that we use an equivalent wheel speed in terms of m/s here.
(3) Initial condition is: $(30m/s, 30m/s)$.
(4) Particle filter is used for sensor noise filtering.
(5) ABS control task period: 20 ms; deadline 20 ms; WCET: 1 ms
(6) Particle filter task period: 2 ms; deadline 2 ms; WCET: 1 ms

Control Algorithm: Based on the Dugoff tire model, an optimal value of the *slip ratio*, determined by the vehicle speed v_v and wheel speed v_w using the equation $slip = 1 - \frac{v_w}{v_v}$, would give the maximum possible friction force during braking. This optimal point depends on the tires used and is typically within the range of 0.1 to 0.25; this information is available offline. Many algorithms exist to control the brake pressure to remain within this optimal region. We experimented with the canonical PID control algorithm [Wittenmark 2011] for the simulation. We used the difference between the optimal point 0.2 and the actual slip ratio as the cost function to estimate the worst case wrong control input.
Safety Space Constraints: The vehicle's SSC was generated by modeling the vehicle as a point mass and using the standard Dugoff tire model [Dugoff et al. 1969]. Under the assumption that the vehicle mass is 1,500 kg, the initial velocity is 30 m/s and the safe stop distance is 55 meters, the generated SSC is shown in Figures 10a and 11a for two road conditions: dry and wet, respectively. These two figures show that as long as the slip ratio is within the preferred region, it is safe and we can guarantee a safe stop distance. Note that with different initial conditions and safe stop distance requirements, the SSC region will be different. How to accurately determine the SSC is beyond the scope of this paper, and is the responsibility of the particular domain specialist.
Simulation Results: Figure 9 shows the simulation results for two different road friction conditions. Note how well the Particle filter tracks the system states, and also note the increase in the total time for a complete stop for a worse road condition.

Since this is a 2-dimensional system, a scatter plot with the corresponding decision boundary determined from the machine learning classification step can be easily plotted. AdaFT generates the sub-spaces shown in Figures 10b and 11b. Note the gains from using adaptive fault-tolerance: almost 50% of the time, the vehicle is in the S_1 sub-space, meaning that the computational workload is significantly reduced.

	LR	*NN*	*SVM*
Number of Trained Parameters	15	93	138
Testing Accuracy	100%	100%	100%

Table II. Comparison of Learning Algorithms for ABS

Using machine learning the sub-spaces were expressed in a lookup table. Since this system only has two dimensions, and from Figure 10(b) and Figure 11(b) there are clear linear boundaries between the sub-spaces, even a linear machine learning model such as logistic regression can perform very well. We include results from three schemes: support vector machine with a Gaussian (rbf) kernel, logistic regression and neural network. The corresponding lookup tables had 138, 15, and 93 trained parameters, respectively. All three algorithms achieved 100% testing accuracy. The precision and recall can both be 100% when using the same probability threshold that tests the testing accuracy. A comparison is shown in Table II and again the number of trained parameters are similar in terms of memory space; and the testing accuracy is 100% for each algorithm.

To calculate the TAAF, we experimented with similar power consumptions as in the first case study: 25 and 50 Watts. Figures 10c and 11c show the improvement in TAAF. Note the considerable improvement from using adaptive fault-tolerance over the standard redundancy-every-time approach.

6.3. Case Study 3: Automated Highway System (AHS) Platoon

Grouping vehicles into platoons is an approach to increase the throughput of road networks [Bergenheim et al. 2012]. Platoons decrease the distances between cars using coupled control approaches, which coordinate acceleration and braking among cars. Such synchronization allows for a considerable increase in traffic throughput. An overview can be found in [Bergenheim et al. 2012]. Cars in a platoon are an example of a distributed CPS, with multiple, cooperating, participants.

State Equations: As seen before, AdaFT requires the user to provide functions to update the controlled plant state and to generate the control signal. The physical model here is much more complex than for the straight line ABS system. Therefore, Carsim [Simulation 2015] integrated with Matlab/Simulink were used for this purpose. Carsim, a commercial automotive simulation tool, provides realistic modeling of vehicles in a variety of road conditions. Matlab/Simulink is used to generate the control signals and update the vehicle state appropriately. Carsim allows the user to configure multiple cars interacting with each other. The user can also specify the physical parameters of the vehicle body, and Carsim would generate the differential equations based on these parameters.

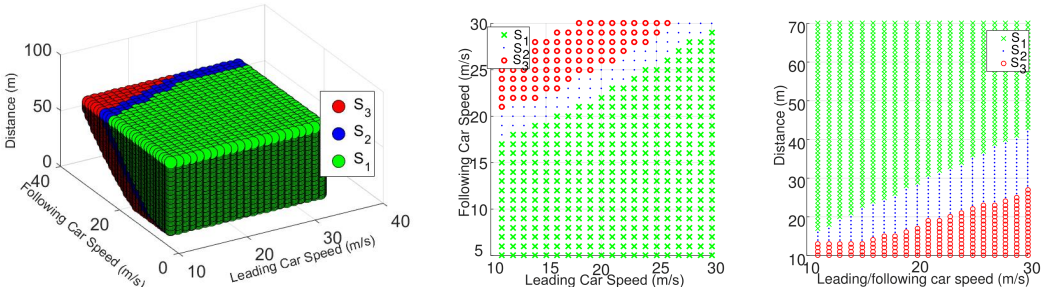
Control Algorithms: The platoon consists of a leader car and one or more followers. The state space vector of the platoon is $(v_\ell, \mathbf{v}_f, \mathbf{s}_f)$ where v_ℓ is the velocity of the platoon leader, $\mathbf{v}_f$ is the vector of velocities of the follower cars, and $\mathbf{s}_f$ is the vector of spacings between each follower and the car in front of it.

The control task is the adaptive cruise control with the constant time-gap policy [Rajamani 2011]. The basic idea is to set a desired inter-vehicle spacing based on this constant time gap, e.g., 2 seconds. This desired distance then varies linearly with the relative velocity. The desired acceleration is computed by the formula: $\ddot{x}_{desired}(t) = -\frac{1}{h}(\dot{\epsilon}(t) + \beta(\epsilon(t) + h\dot{x}(t)))$, where h is a design parameter, β is the desired time gap, and ϵ is the inter-car distance. Details about this algorithm can be found in [Rajamani 2011]. Obviously, a negative acceleration means braking.

Safety Space Constraints: The minimum inter-car distance is set to 5 meters; anything less is defined as control failure and unsafe condition.

	LR	*NN*	*SVM*
Trained Parameters Size	15	153	788
Training Accuracy	78.56%	99.58%	99.62%

Table III. Comparison of Learning Algorithms for Platoon



(a) Sub-Spaces (b) Distance 40 Meters (c) Same Speed

Fig. 12. Platoon Follower Car Sub-Spaces

Simulation Results: Our results have shown that the logistic regression classification scheme performed poorly, while more complex algorithms did much better, as shown in Table III. After adjusting the prediction thresholds, the achieved precision is close to 100%. The plots of thermal damage to the processors are similar to those in Case 2 and are, therefore, omitted.

We use a different perspective to analyze the results here. Figure 12a shows the sub-spaces for the first follower car. Figure 12b shows how to deal with the follower car joining the platoon and synchronizing its speed appropriately. Sub-Spaces are shown for various speeds of the joining car for an inter-car distance of 40 meters. If its speed is low and it needs to accelerate to join the platoon, then little, if any, fault-tolerance is needed. If its speed is higher, the amount of fault-tolerance required increases. In Figure 12c, we show the impact of inter-car distance and platoon speed on the sub-spaces after speed synchronization has been established between the leader and the follower vehicles. As the inter-car distance decreases and/or the platoon speed increases, the highway throughput increases (in terms of cars per time unit crossing a given point) but the computational burden also increases in each car, since a higher level of fault-tolerance is needed. For example, at a speed of 25 meters/sec, an inter-car distance of less than 21 meters will require fault-correction; between 22 and 34 meters, fault-detection is sufficient, and above 34 meters, no fault-tolerance is needed.

7. RELATED WORK

The idea of adaptive fault tolerance is not new; it goes back more than two decades. In [Goldberg et al. 1993], the authors pointed out that the fault tolerance needs of an application, and the fault tolerance capabilities of the micro-controller could change as time goes by, therefore an adaptive technique was presented. In [Fraga et al. 2003], the authors proposed an object oriented way to manage adaptive fault tolerance. The fault tolerance management unit is informed about the reliability requirements of the application; it then adjusts the level of fault tolerance to suit the reliability requirements.

Considerable recent work also contributed to the area of adaptive fault tolerance, due to the fact that increasingly more safety critical systems are nowadays controlled by embedded controllers. In [Liu et al. 2008], an On-demand Real-TimEGuArd (ORTEGA) was proposed to allow for efficient resource utilization based on the runtime state space. The idea is to divide the state space of the controlled plant into two subsystems: a high-assurance-control (HAC) and a high-performance-control (HPC) subsystem. They introduced a two-level FT option to be adjusted in real-time.

Most real-time fault-tolerance research assumes that faults cause failures at the application level. In [Song et al. 2013], Song, *et al*. explained why system-level software (e.g., RTOS scheduling, memory management and I/O processing) is closely tied to failures; indeed, 65% of hardware errors would corrupt OS state [Li et al. 2008]. They claimed that such a situation has received little attention, and they proposed Computational Crash Cart (C^3), with the main idea of dividing the system components based on their functionality (i.e., scheduler, I/O). After a fault is detected, the faulty component can perform a focused micro-reboot, rather than having to restart the whole system. Later they extended this work in [Song and Parmer 2015], to allow the system to have a run-time monitoring as well as validation capability.

Swetha, *et al* introduced the *Enhanced Resource Management Scheme* (ERMS) and compared it with the traditional redundancy approach [Swetha et al. 2014]. A new scheduling method, a combination of dynamic planning and dynamic best effort approach, allowed for joint scheduling of periodic and aperiodic tasks which also include online reconfiguration for error management. This fault recovery technique allows all critical tasks to meet their deadlines and the system continues functioning at a minimal safe functionality upon errors. This scheme has been analyzed and evaluated, using simulation, on an automotive cruise control system.

Bak, *et al.* developed a combined online/offline approach [Bak et al. 2014] for the well known Simplex architecture. Their approach uses aspects of a real-time reachability computation, which also maintains safety, but is less conservative. The switch logic for their Simplex architecture will, like the traditional Simplex architecture, guarantee that the system never enters an unsafe state (safety), but uses the complex controller as much as possible (minimize conservatism). Simplex mainly deals with software fault tolerance, with the assumption that the simple controller is formally verified without any software bugs, and that the complex controller gives potentially better quality of control but without a formal verification. Since only one copy of each controller version is deployed during run time, no hardware fault tolerance is supported. One possible combination of our work and the Simplex architecture is to introduce a more flexible and more powerful fault tolerance hierarchy (both software and hardware): with the knowledge of the physical side state information, the system might be able to determine which version of the control task and how many copies of the control tasks to run at a specific time.

The paper by Bogdan and Marculescu [Bogdan and Marculescu 2011] argues that one can expect many cyber-physical computational workloads to show fractal behavior. They point out that if this is the case, it will affect computational resource allocation. As far as AdaFT is concerned, what is of relevance is the computational burden as a function of the application (controlled plant) physical states. In its current form, the framework accepts traditional digital control workloads. As new workload formulations emerge from the CPS community, they can be included in the AdaFT framework.

Note that most prior work on adaptive fault tolerance focused on the cyber side of CPS. The cyber system was either informed about the current state by the physical plant and the required fault tolerance level, or these conditions were assumed to be given. By contrast, our work uses a cyber and plant side co-design approach to provide

in real-time the physical side state information to the cyber side, based on which the cyber fault-tolerance level can be determined.

8. CONCLUSION

As cyber-physical systems become ever more complicated, a trend has emerged towards running the control tasks on an integrated computation platform rather than on isolated controllers. Traditionally, massive always-on redundancy has been used to ensure reliable controller performance. However, in many, if not most, instances, the controlled plant is so deep within its allowed state space that occasional controller errors do not cause controller plant failure. This motivates an adaptive approach to fault-tolerance. Such an approach significantly reduces the computational burden of the controller. This reduced burden, in turn, leads to lower controller operating temperatures which enhances processor lifetime.

This paper describes an generalized software simulation framework, AdaFT, for an adaptive fault tolerance in CPS. AdaFT first partitions the state space of the controlled plant based on how much fault-tolerance is required. Then a machine-learning based approach is used to generate a compact memory look-up table indicating the required level of fault tolerance. When provided with power consumption information, the framework carries out a thermal analysis of the cyber elements and uses that information to estimate reliability. Our current work with respect to possible extensions to AdaFT include: range of control systems being evaluated, i.e., effective handling of nonlinear and large systems; hierarchical management of distributed applications with multiple, interacting, centers of control, automatic load tuning/balancing, and hardware provisioning according to a user-defined expected system reliability or lifetime.

The AdaFT framework can be used in every stage of the design process. It can be used to determine the impact of control task dispatch frequencies on system performance and reliability. Given processor failure rates, its calculation of thermally accelerated aging can determine hardware provisioning for specified operational lifetimes. Its analysis of controlled plant dynamics can be used to set the appropriate level of fault-tolerance required for safe plant functioning. At a time when CPS complexity is increasing and demands on reliability are increasing, AdaFT facilitates the use of adaptive fault tolerance that allows for an economical and safe management of resources in cyber-physical systems.

REFERENCES

S. Bak, T.T. Johnson, M. Caccamo, and L. Sha. 2014. Real-time reachability for verified simplex design. *Real-Time Systems Symposium (RTSS)* (2014).

C. Bergenheim, S. Shladover, and E. Coelingh. 2012. Overview of platooning systems. *Proceedings of the 19th ITS World Congress* (2012).

P. Bogdan and R. Marculescu. 2011. Towards a science of cyber-physical systems design. In *Cyber-Physical Systems (ICCPS), 2011 IEEE/ACM International Conference on*. IEEE, 99–108.

L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

A. Burns and R.I. Davis. 2015. Mixed criticality systems – A review. *IEEE Real-Time Systems Symposium (RTSS)* (2015). www-users.cs.york.ac.uk/burns/review.pdf

J. Clausen. 1999. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen* (1999), 1–30.

J. Cooling. 2013. *Real-time Operating Systems*. Lindentree Associates.

P. N. Currier. 2011. A Method for Modeling and Prediction of Ground Vehicle Dynamics and Stability in Autonomous Systems. *PhD Thesis in Virginia Tech University* (2011).

H. Dugoff, P. S. Fancher, and L. Segel. 1969. Tire performance characteristics affecting vehicle response to steering and braking control inputs. *Transportation Research Board* (1969).

L. Escobar and W. Meeker. 2006. A review of accelerated test models. *Statist. Sci.* (2006), 552–577.

J. Fraga, F. Siqueira, and F. Favarim. 2003. An adaptive fault-tolerant component model. *IEEE International Workshop on Object-Oriented Real-Time Dependable Systems* (2003).

J. Goldberg, I. Greenberg, and T.F. Lawrence. 1993. Adaptive fault tolerance. *IEEE Workshop on Advances in Parallel and Distributed Systems* (1993).

A. Goyal and A. N. Tantawi. 1987. Evaluation of performability for degradable computer systems. *IEEE Trans. Comput.* 100 (1987).

I. Koren and C. M. Krishna. 2007. Fault-tolerant systems. *Morgan Kaufmann* (2007).

C.M. Krishna. 2015. Ameliorating Thermally Acclerated Aging with State-Based Application of Fault-Tolerance in Cyber-Physical Computers. *IEEE Transactions on Reliability* 64 (2015).

C.M. Krishna and I. Koren. 2013. Adaptive Fault-Tolerance for Cyber-Physical Systems. *CPS Workshop* (2013).

C.M. Krishna and K.G. Shin. 1987. Performance measures for control computers. *IEEE Trans. Automat. Control* 32 (1987).

J. Lehoczky, L. Sha, and Y. Ding. 1989. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.* IEEE, 166–171.

M. Li, P. Ramachandran, S. Kumar Sahoo, S. V Adve, V. S Adve, and Y. Zhou. 2008. Understanding the propagation of hard errors to software and implications for resilient system design. *ACM SIGARCH Computer Architecture News* (2008).

X. Liu, Q. Wang, S. Gopalakrishnan, W. He, L. Sha, H. Ding, and K. Lee. 2008. ORTEGA: An efficient and flexible online fault tolerance architecture for real-time control systems. *IEEE Transactions on Industrial Informatics* (2008).

J. F. Meyer. 1982. Closed-form solutions of performability. *IEEE Trans. Comput.* (1982).

J. F. Meyer, D. G. Furchtgott, and L. T. Wu. 1980. Performability evaluation of the SIFT computer. *IEEE Trans. Comput.* (1980).

University Michigan. 2012. Matlab and Simulink Tutorial. (2012). http://ctms.engin.umich.edu/CTMS/

R. Moazzami, J. C Lee, and C. Hu. 1989. Temperature acceleration of time-dependent dielectric breakdown. *IEEE Transactions on Electron Devices* (1989).

K. P. Murphy. 2013. Machine Learning: a Probabilistic Perspective. *MIT Press* (2013).

R. Rajamani. 2011. Vehicle dynamics and control. *Springer* (2011).

J. Schoen. 1980. A model of electromigration failure under pulsed condition. *Journal of Applied Physics* (1980).

D. K Schroder. 2007. Negative bias temperature instability: What do we understand? *Microelectronics Reliability* (2007).

K.G. Shin, C.M. Krishna, and Y.-H. Lee. 1985. A unified method for evaluating real-time computers and its application. *IEEE Trans. Automat. Control* (1985).

Mechanical Simulation. 2015. CarSim. (2015). http://www.carsim.com/

K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and David Tarjan. 2004. Temperature-aware microarchitecture: Modeling and implementation. *ACM Transactions on Architecture and Code Optimization (TACO)* (2004).

J. Song and G. Parmer. 2015. CMON: a Predictable Monitoring Infrastructure for System-Level Latent Fault Detection and Recovery. *Real-Time and Embedded Technology and Application Symposium (RTAS)* (2015).

J. Song, J. Wittrock, and G. Parme. 2013. Predictable, Efficient System-Level Fault Tolerance in Cˆ 3. *Real-Time Systems Symposium (RTSS)* (2013).

A. Swetha, R. Pillay V, and S. Punnekkat. 2014. Design, Analysis and Implementation of Improved Adaptive Fault Tolerant Model for Cruise Control Multiprocessor System. *International Journal of Computer Applications (IJCA)* (2014).

S. Thrun, W. Burgard, and D. Fox. 2005. *Probabilistic robotics*. MIT press.

S. Vestal. 2007. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. *IEEE Real-Time Systems Symposium (RTSS)* (2007).

W. J. Vigrass. 2004. Calculation of semiconductor failure rates. *Harris Semiconductor* (2004).

B. Wittenmark. 2011. Computer-controlled systems: theory and design. *Courier Dover Publications* (2011).