# Effects of Computation Speed on the Stability of a Self-Controlling Process

Nilgun Fescioglu-Unver  Mieczyslaw M. Kokar
*Northeastern University*
*{nunver, kokar}@coe.neu.edu*

## Abstract

*In this paper we consider the impact of the speed of the processor on the stability of the adaptation process. The problem is investigated on a case study that involves a real-time system for monitoring illuminations of multiple emitters by one receiver mounted on a moving platform. This is a self-controlling system whose goal is to schedule the receiver so that the uncertainty about the times of illuminations of radars in the direction of the platform is minimized. The system has several controllers and a scheduler which are using the same computer as the computation resource. This paper demonstrates the effects of computation time on a real time self-controlling system and shows the changes in stability of the system under high computation loads.*

## 1. Introduction

The paradigm of self-controlling software [1,2] is becoming recognized as a way of achieving both better performance and robustness in the presence of unexpected changes in the environment. Self-controlling software changes its behavior in response to external feedback it receives during operation which comes from human users or external world that the software system operates in. Some of the ways to change the behavior of the system are adjusting some parameters or changing the structure. We are pursuing a control theory based approach to self-controlling software where the software is treated as a plant. The plant is viewed as a dynamical system with dynamics related to both the external environment and the computing system.

Self-controlling software is also a part of autonomic computing [3]. Autonomic computing deals with computing systems that can manage themselves while trying to acquire high level objectives of administrators. Self-management of a system includes self-configuration, self- optimization, self-healing and self-protection. For dealing with changing run-time conditions, use of control theoretic approaches has been suggested by various authors, e.g., Kephard and Chess [3].

Self-control consumes two kinds of resource: time and CPU. While we normally refer to the CPU resource as "CPU time", this resource is quite different from the physical time resource. In order to analyze behaviors of self-controlling software, such software must be considered in the context of its operation. A representation of such context that captures dependencies between CPU time, physical time and other things is shown in Figure 1.

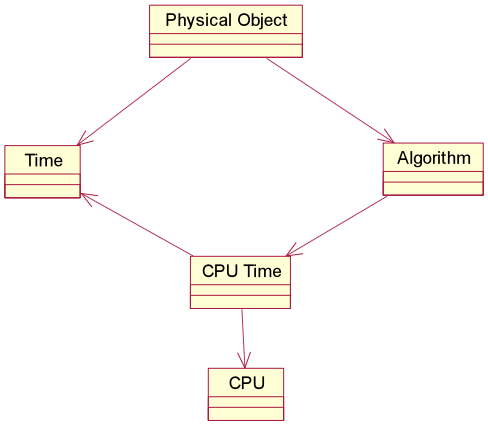


**Figure 1: Context and Dependencies for Self-Controlling Software**

This diagram shows (in UML notation) five concepts: *Physical Object, Time, Algorithm, CPU Time* and *CPU*. Physical Object is in this picture since we assume that software interacts with the physical world. The behavior (or state) of a Physical Object depends on both the passage of time (we assume this is a dynamical system) and the activations it receives from an external activator – we represent it here as *Algorithm*. In this paper we assume that the algorithm is a Self-Controlling System, i.e., it both affects the Physical Object and controls itself. The

Algorithm depends on the CPU time, i.e., the speed of the CPU multiplied by the time unit. Thus CPU Time depends on Physical Time, too. For completeness, we show CPU, which is a physical device. The parameter of CPU Speed is associated with this object.

These dependencies imply that various elements in the context of operation of a self-controlling system may have conflicting demands for resources, in this case CPU Time and Time. In a situation where the computer makes decisions for a physical system, the time that the computer is spending for the derivation of the decision may affect the physical system, especially when the decision is derived too late. Self-control decisions consume CPU Time due to both algorithm selection (another algorithm does that) and the execution of the algorithm. (Both algorithms are modeled here as Algorithm.) Since the adaptability of the system depends on the timeliness of decisions for the Physical Object, the delays in making control decisions, both self-control and activation of the Physical Object, may affect the adaptability metric.

In this paper we present some of the results of our study of adaptability of a self-controlling system. Since, as shown in Figure 1, CPU Time is one of the components that may affect adaptability, we had to implement a system in which this component is a first-class object, i.e., such an object that can have its own properties and that can be manipulated by the simulation. Then the effects of changes in this object can be measured. We used Ptolemy, a tool with a component assembly framework, to achieve this objective. In our implementation, we were able to vary the speed of the CPU and then observe the impact of such changes on the performance (adaptability) of the system. The varying of the CPU speed simulates the effect of using algorithms with higher time-complexity. We were able to not only provide support for the claim that CPU speed impacts the adaptability characteristic, but also captured quantitative relationships between the CPU speed and the adaptability.

## 2. Motivation

The physical object studied in this problem is a radar-receiver system [4]. The goal of the system is to manage a receiver whose goal is to monitor multiple radars operating in the environment surrounding the receiver. Each radar operates in a different frequency band. So the receiver can "see" the radar only when it tunes to the frequency of a given radar. The time when a receiver is tuned to a particular radar's frequency is called a *dwell*. Radars illuminate the receiver once per rotation. For instance, a typical radar has a rotation period of 4 seconds and thus would illuminate the receiver roughly every 4 seconds. The length of the illumination time interval (of the dwell) is rather short, typically 0.5% of the rotation period. The goal of the system is then to schedule the dwells in such a way that they coincide with the times of illuminations by particular radars in the direction of the platform.

The approach used in this paper is called the control theory metaphor of software development [1]. In this paradigm, the control goal (the controlled output variable) must be represented by a quantitative metric. In control literature it is called the *set point*. In this system the objective is to minimize the uncertainty about particular illuminations. This means that the system wants to know at what times the radars are looking at the platform. Knowing does not necessarily imply measuring. In other words, the system has two choices – either tune in to a specific radar's frequency band or use estimation (tracking). The uncertainty about a system is measured by the entropy and the system's controlled output variable then is simply the entropy.

Another possible controlled output variable of the system could be defined. This variable is often called the Quality of Service (QoS). For instance, for the application discussed in this paper, this variable would have to be defined in such a way that when a radar is not being observed, its controller level QoS would increase; when it is observed, the QoS would decrease. In Autonomic Systems utility functions provide the objective function for self-optimization [5].

The problem addressed in this paper is an instance of the so called *non-preemptive scheduling problem*, which is known to be NP-complete [6]. One possible approach to solve this problem is to develop the schedule off-line, before the flight, i.e., before the mission of the platform. For the application discussed in this paper, this approach is referred to as *Fixed Scan Scheduling* [7]. This technique guarantees detection of each illumination at low loads, i.e., for a small number of radars. But when the system load increases, it cannot adjust itself. A self-optimization approach would make the system adaptable. In the self-adapting case, instead of giving the same importance to all emitters, the emitters that have not been detected as successfully have higher priority as compared to those emitters that are detected more frequently.

To study the problem described in this paper we developed a simulation using Ptolemy II [8]. Ptolemy II is a Java-based component assembly framework with a graphical user interface, developed at UC Berkeley. The Ptolemy project studies modeling, design and simulation of concurrent, real time, and embedded systems. It is published freely and is open source. For our simulations we used the Ptolemy II Version 3.0.2. Ptolemy has different modes of computation including Discrete Event, Finite State Machine and Timed Multitasking models, and different calculation models can be used in the same system hierarchically.

In this paper we will demonstrate how the simulation was used to show how the system tries to self-optimize using the control architecture approach. The effects of computation time on the system will be investigated and the instability due to the delays in computation will be demonstrated.

The remainder of the paper is organized as follows: In Section 3, the detailed description of the system is presented. In Section 4, the system implementation in Ptolemy II is described. In Section 5, we present and discuss the simulation results with low and high computation times. Finally, in Section 6, the findings are summarized and future research directions are discussed.

## 3. System Description

In this section we will describe the self controlling system and its components.

### 3.1 System Components

The physical system is composed of a number of emitters, a receiver and the platform. The computer system includes emitter level controllers, a scheduler and a computer. The computer system is a resource management system, whose resource is the receiver (physical) time. Since the computer system also controls itself, one more resource is considered – Computer.

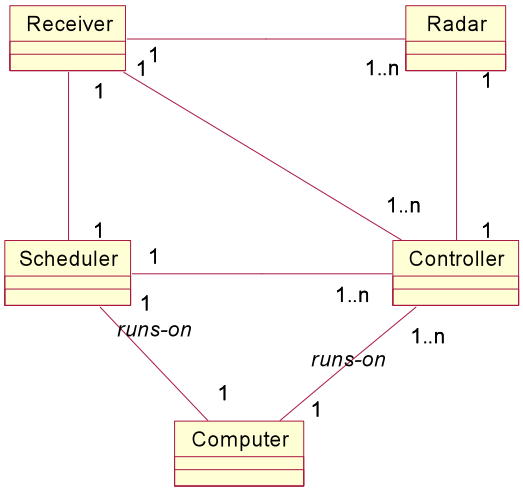


**Figure 2. UML diagram of the resource management system**

Figure 2 represents the UML class diagram for the whole system. Each Radar in the environment is associated with the Receiver. Receiver tries to keep track of all Radars. For each Radar, there is a Controller in the system and all Controllers are associated with both the Receiver and Scheduler. Finally Scheduler and Controllers are associated with the Computer and use it as the computation resource.

In the simulation it was assumed that there are 10 Radars in the environment. Each of the Radars rotates continuously. However, a Radar illuminates only in the direction it is pointing to at a particular time. We model radars as timed automata. Each Radar has a different beam width. For instance, a Radar that has a beam width of 2° can point in 180 different directions. It points in a direction for the time interval *dt* then switches to the next state. The state transition diagram for a Radar is shown in Figure 3.

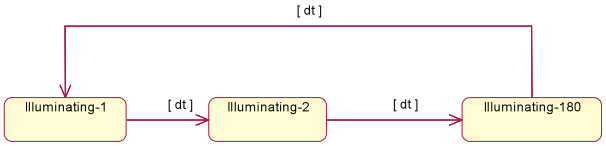


**Figure 3. The behavior of the radars**

The Receiver is controlled by the Scheduler. The Scheduler generates scheduling events (*sched-1,..., sched-n*). The Receiver, in response to an event *sched-i* switches to the state *PointingAtRadar-i* for the duration of the dwell time prescribed by the Scheduler. When pointing at a particular Radar, the Receiver either detects the Radar or not, depending on whether the Receiver is currently located on the direction in which the Radar is illuminating during this dwell time or not (see Figure 4). If the Receiver is on that line, the Receiver generates an event of detection, i.e., *detect-i* is set to 1. Otherwise *detect-i* is set to 0. Since the Receiver is on a moving platform, the line with respect to a particular Radar changes constantly. Thus even if the Receiver is on the line of a particular Radar at the beginning of this dwell, it may be outside of the line at some point in the middle of the dwell. Consequently if this happens, the Radar cannot be detected. For this reason *detect-i* event is set to 1 only when the Receiver is within the line of view of the Radar for the whole dwell period.

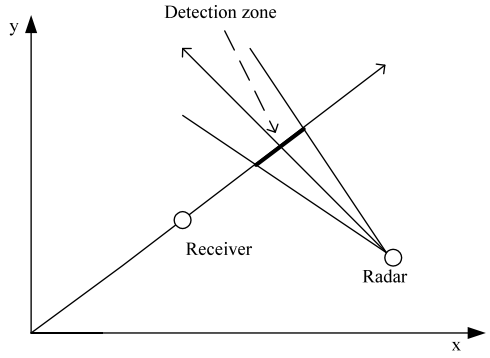

**Figure 4. The detection scenario**

The Receiver is modeled as a timed automaton as in Figure 5. Its continuous dynamics is given by the equation

$$\frac{dV}{dt} = 0$$

where $V(0)=V_0$. In other words, the receiver is moving with a constant velocity $V_0$.

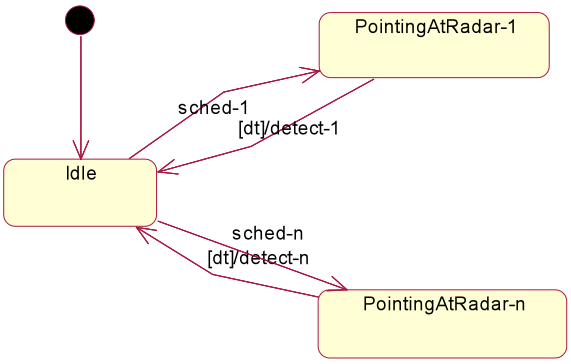


**Figure 5. The behavior of the receiver**

The detection event is passed to an appropriate Controller. In this system there is one Controller for each Radar. In response to a detection (or non-detection) event the Controller needs to update the uncertainty information of the Radar by calculating its entropy. Controller keeps track of the probability distribution for the beam direction the Radar is currently emitting at. The probability distribution is updated at a detection, no-detection or no-observation event (if the Radar is not looked at for a specified period). Entropy of the Radar is calculated by using this probability distribution. After the Controller updates the entropy information, it needs to compute the time at which the Receiver should be pointed again at that Radar. To perform these calculations the Controller needs the Computer. For a calculation, Controller sends a calculation request to the Computer and waits for the answer Computer returns. The time of the return of the result to the Controller depends on the CPU speed of the Computer and awaiting computation requests in the Computer's queue. After Controller receives the result of the next dwell time, it sends the control description word (CDW) together with the entropy information to the scheduler. The CDW specifies when and how long the receiver should look at a specific radar.

After the Scheduler receives the dwell-i event, which indicates that the receiver is idle and is waiting for a new schedule, it re-computes its schedule. Scheduler has a queue of received CDWs from the Controllers. Each CDW contains information about the radar id, the start time and the duration of the dwell. In addition to the CDW, Controllers send information about detection probability and entropy of the emitter to the Scheduler. Scheduler selects the CDW taking into consideration this dwell's probability of detection and entropy of the radar. To add an incoming CDW to its queue and to compute the schedule, the Scheduler needs the Computer. After the schedule is computed, the CDW information is sent to the Receiver.

The last system component is the Computer. The Computer is the single computation resource of the system and can answer a single computation request at a time. It receives calculation requests from all Controllers and the Scheduler. It services the requests with higher priority given to the Scheduler, and in a first-come first-served mode among the Controller requests. The Computer is modeled as a separate entity to investigate the impact of the speed of the processor on the stability of the adaptation process.

### 3.2. Self-Control of the System

The Controllers are the major elements in the adaptation process. In this experiment we used a simple version of Controller, i.e., a P (or *proportional*) control law. Each Controller keeps track of the related Radar's detections – non-detections and the uncertainty, represented by the entropy. The control based sensor management architecture can be seen in Figure 6.

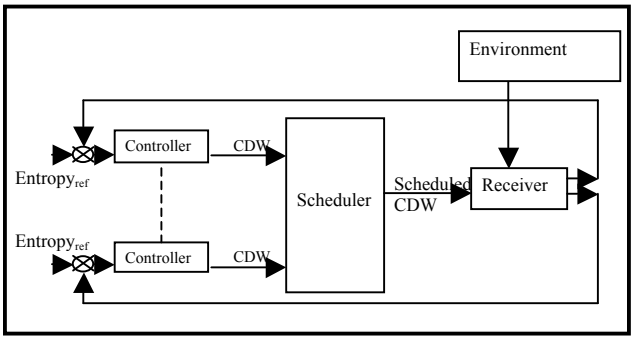


**Figure 6. Control based sensor management architecture**

The responsibility of the Controller is to generate the CDWs. The start of the next dwell is a component of a CDW that needs to be re-computed for each dwell. If the illuminations of the Radars were perfectly periodic, if the Receiver was stationary and if the distance between the emitter and the platform were known, then it would be possible to predict the exact time of the next dwell. The next dwell time would be last dwell time ($t_{ds}$) plus the revisit time ($\tau_{rv}$). In our case, in order to compensate for the phase shift, $\tau_{rv}$ is adjusted such that:

$$\tau_{rv}(i,t) = \bar{\delta}(i,t)^2 . T_{eip}(i)$$

where $T_{eip}$ is the illumination period of the radar,

$$\bar{\delta}(i,t) = \begin{cases} 1, & \text{if } \delta(i,t) > 1 \\ \sqrt{\dfrac{T_{eit}(i)}{T_{eip}(i)}}, & \text{if } \delta(i,t) < \sqrt{\dfrac{T_{eit}(i)}{T_{eip}(i)}} \\ \delta(i,t), & \text{otherwise} \end{cases}$$

where $T_{eit}$ is the illumination time (duration of a single illumination) and

$$\delta(i,t) = \varepsilon \cdot K(i)$$

where K(i) is the proportional controller parameter and

$$\varepsilon = Entropy_{ref}(i) - Entropy(i)$$

ε is the difference between the reference input, which is the maximum entropy of the Radar and the entropy computed by the Controller.

The effect of the above Controller can be summarized as follows. If a CDW is not scheduled, the uncertainty, hence the entropy of the Radar, gets higher. When entropy gets higher, the difference between the maximum entropy and current entropy (ε) gets smaller, which results in a smaller δ. When δ gets smaller, the revisit time gets smaller, which means more CDWs will be generated and the probability of detection of an illumination will increase. If a dwell on an emitter results in detection, the entropy gets smaller, the difference between maximum entropy and current entropy (ε) gets larger and so δ gets larger. When δ gets larger, the revisit time gets larger and fewer CDWs are generated.

Each Controller calculates a CDW and sends the request to the Computer. Detection of a Radar decreases its entropy whereas no-observation of it (when a specified amount of time passes without the radar is looked at) increases the entropy. A detection event decreases the entropy to its minimum value since in such a case we know for certain that the Radar is illuminating at the Receiver. No-observation of the radar increases the entropy because as time passes if the radar is not observed, the uncertainty about it increases.

CDWs generated by the Controllers have an expiration time associated with them. As the key element of the CDW is the time of next dwell, if the CDW is not scheduled until the calculated $t_{ds}$ , it expires and a new CDW needs to be calculated for the Radar. Expiration of the CDWs increases the need for a Computer with a fast processor. Computer's speed affects the expiration frequency of the CDWs if the expirations are due to the delays in computation. Our aim is to simulate the system with fast and slow processor Computers and investigate the behavior and the stability of the system.

## 4. Implementation of the System Using Ptolemy II

Ptolemy II is a modeling and simulation environment that supports heterogeneous, concurrent modeling and design. It is composed of a set of Java packages and a graphical user interface.

Ptolemy II software environment implements an actor-oriented design methodology. An actor is defined as an encapsulation of parameterized actions performed on input data to produce output data [9]. By using the actor orientation, functionality concerns are separated from the component interaction concerns. The component interaction and control flow between actors is done through frameworks. In the design of this simulation different frameworks were used hierarchically. At the top level, discrete-event (DE) domain is used. In the discrete-event model, actors share a global time and communicate through events. The global notion of time makes it possible to investigate the effect of communication delays and timing behaviors of the system [10]. In the next layers finite state machine (FSM) and timed multitasking (TM) domains are used. The Timed Multitasking domain is an event triggered model; it controls the time at which outputs are produced. It can effectively control starting and stopping times of tasks and so obtains deterministic timing properties [11]. The Timed Multitasking domain is used to model the computer's behavior. By using this domain in computer, the duration of the calculations is controlled and changes in the stability of the system due to the processor speed are observed.

The first layer of the system is shown in Figure 7. In the first layer, all the entities are shown as separate actors and communication between actors is done via their ports. In this layer the control flow through actors is done in discrete event mode. As all entities share the same global time and they are communicating through events, the Discrete Event domain was the best fit for the scenario. The inter-connections among Receiver, Radar-control (Radar-control actor holds n-controllers in it, where n is the number of radars), the Scheduler and the Computer entities are shown in the first layer. The Environment and Detect-calculator entities are supporting entities of the system, where the Environment contains n-radars and Detect-calculator calculates whether detection has occurred or not, given the information from receiver and environment. Detect-calculator just calculates whether the physical factors at the moment will generate a detection event or not (whether radar is illuminating in the direction of the receiver, whether it is within the detection range, etc.). All the actors seen in the first layer are composed of other actors.

The Computer behavior is modeled in the Timed Multitasking (TM) domain. The TM domain gives the Computer entity the capability to control the duration of the calculations, give the priority to requests coming from specified entities and choose a scheduling strategy (preemptive or non-preemptive). In this problem, the Computer schedules the computations in non-preemptive mode and gives higher priority to Scheduler requests.

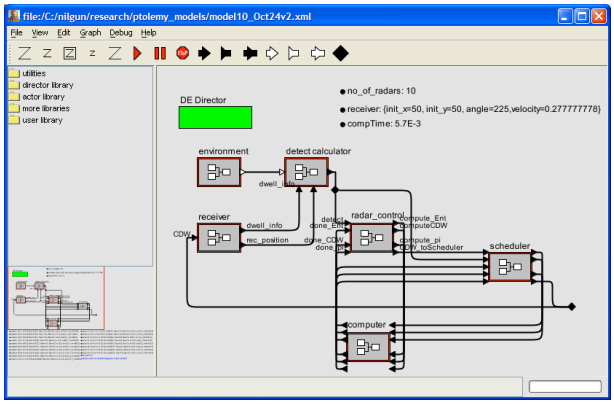


**Figure 7. First layer in system model**

The use of Ptolemy II in modeling and simulating this system made it straightforward to visualize the behavior of the Computer as a separate entity. The control logic and the entity relationships are implemented without difficulty by using Ptolemy's different modeling domains.

## 5. Simulation Results

One of the goals of this research is to investigate the behavior of the system in response to the changes in the time spent for computing sensor scheduling decisions. Although a full investigation would involve the analysis of the computation duration for each separate algorithm within the system, so far we limited our study to the case where the computer speed changes. This is, in a sense, equivalent to the situation where the speed of computation of all of the algorithms changes at the same rate. The main goal for us was to demonstrate the impact of the processing speed on the stability of the adaptation (self-control) process.

The Computer is a separate entity in the system. It can hold a queue of calculation requests. The system goal is to manage the Receiver as a resource and the Receiver being idle for a time period is a loss for the system. To make the Receiver dwell at some radar for a specified period, the Scheduler must send a CDW to the receiver. The Scheduler is given the highest priority in the calculation queue so that requests from the Scheduler are serviced first by the Computer. Controllers have the second priority and all Controller requests are served in a first-come first-served fashion. Controllers keep track of each Radar's entropy and the average of all Radar entropies gives the overall system entropy. A change in the system entropy with respect to time is compared for three different computers, two with fast processors and one with a slow processor.

In the first simulation, the system is run with a computer entity that has a fast processor. Ptolemy's simulation environment gives the flexibility to assign different calculation times for each computation. For ease of comparison, all calculations are given the same time in this simulation. A single computation request incoming to the Computer from the Scheduler or Controllers is serviced within $5.10^{-4}$seconds. The resulting average entropy vs. time is shown in Figure 8.

It is seen that the average entropy of the system quickly drops from its maximum 2.0 point to a range between 1.7 and 1.3 and stabilizes at that level. Simulation is repeated with a Computer entity that has a faster processor which can do a single computation in $5.10^{-10}$ seconds. As the system load is not heavy in this problem (it is assumed that there are 10 radars in the environment, so 10 controllers and 1 scheduler asking for computation) using a faster Computer does not improve the results. The average entropy of the system with a processor which can answer any computation request in $5.10^{-10}$ is shown in Figure 9.

However, with a higher workload, the use of a faster processor helps the system achieve stability more quickly. To demonstrate the effect of processor speed on system stability, the simulation is repeated with a computer entity that has a slow processor. In this case, a single computation request is answered (this includes just the computation time and not the waiting time) in $5.10^{-3}$ seconds with this Computer. The result is given in Figure 10.

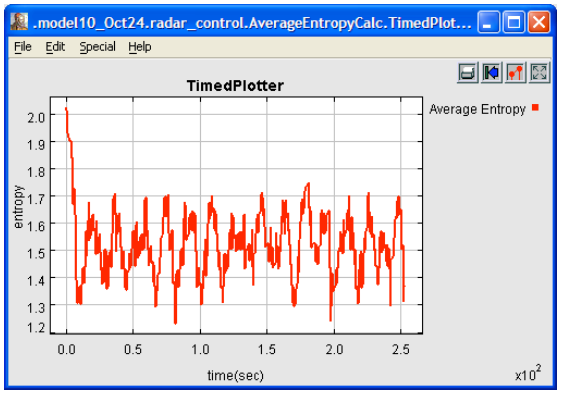


**Figure 8. Average entropy of the system when a fast processor is used**

It can be seen in Figure 10 that the system entropy stays in a high range between 1.8 and 2.0 in the first 100 seconds. In the next 40 seconds, the average entropy drops to a range of 1.8-1.5, and after the first 140 seconds the entropy drops to the 1.75-1.3 range.

The reason of the increase in the average entropy is as follows. Because of the slow processor and high computation times, the computation requests of the controllers and scheduler wait in the computation queue for a long time. As described in Section 2, the CDW's main component that is recalculated each time is $t_{ds}$ –

time to start the next dwell. If the CDW calculation request of the Controller is answered after the current time becomes equal to $t_{ds}$, the Controller receives a CDW that has already expired. The Controller sends the incoming CDW to the Scheduler, but the Scheduler never selects that CDW to send to the Receiver because it is already expired. Following this reasoning one might expect that the average entropy level should always remain high as the Controllers which receive expired CDWs may never catch up. However this is not the case with this plant because the time of the next dwell is calculated with a self-controlling approach. Once the Radars that the Receiver first started to observe become detected, their entropy drops and their CDW computation requests decrease. The decrease in the workload of Computer gives the other CDW computation requests a chance to get computed before they are already expired. After the first 100 seconds, more than half of the radars start to get observed and after 130 seconds the other radars start to be observed as the back to back detection of the already observed Radars decreases the number of CDW computation requests made to the Computer. As the CDW computation requests decrease, the workload of the Computer decreases and the CDW calculations get completed before expiration time. After the first 140 seconds the average entropy drops to the 1.75-1.3 level and the system stabilizes.

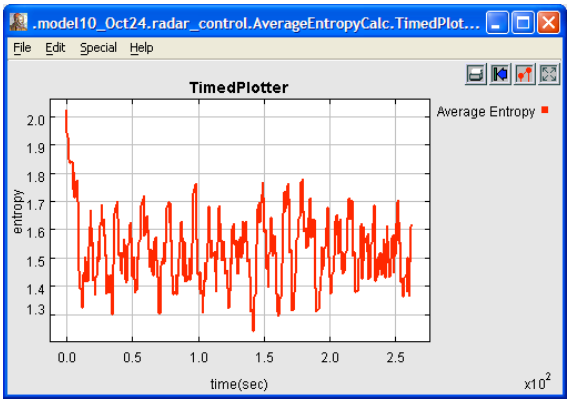


**Figure 9. Average system entropy using a processor which can respond to a calculation request in $5.10^{-10}$ sec.**

The time it takes for the system to stabilize depends on the computation times, and thus it depends on the speed of the computer. With very high computation times the system may never be able to stabilize. Figure 11 shows the change in entropy level of the system with different computation times.

The time the system needs to drop to a certain level of entropy directly changes with the length of the computation time, too. Figure 12 demonstrates the change in the time of stabilization with respect to the computation time. Entropy level of 1.6 is taken as the stabilization level of the system. As the speed of the Computer decreases and consequently computation time increases, the time it takes the system to reach the 1.6 level increases. In systems with computation times longer than 5.5E-3, the system can never reach that level and thus is unstable.

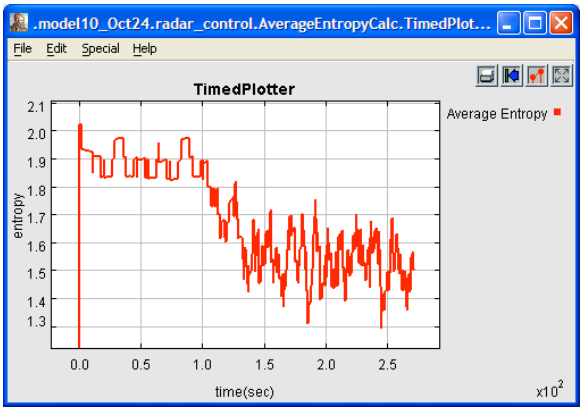


**Figure 10. Average system entropy using a processor which can respond to a calculation request in $5.10^{-3}$ sec.**

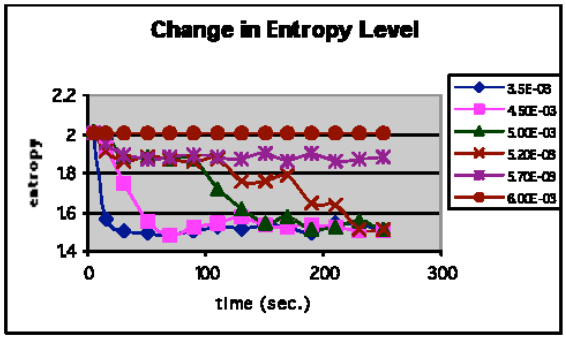


**Figure 11. Change in entropy level with different computation times**

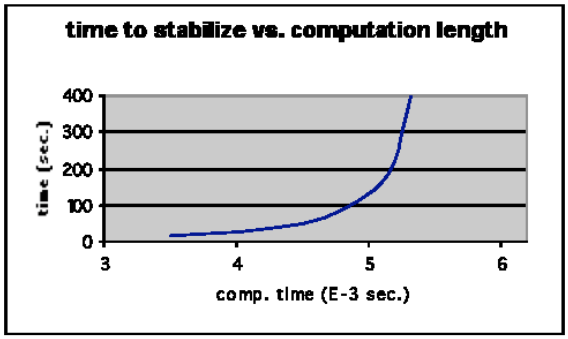


**Figure 12. Time to stabilize vs. computation length**

## 6. Conclusion

In this paper we investigated the impact of the speed of the processor on the stability of a self-controlling system. For the case study we used a physical system whose goal is to schedule a receiver that monitors radar illuminations. The system goal is to minimize the entropy (uncertainty) of the detection decision. The entropy for a specific radar increases when the uncertainty about the time of illuminating in the direction of the receiver increases. The system has a single computation resource - the computer. In our experiments we observed that the stability of the self-controlling system developed for this application highly depends on the computation speed. Simulations were performed for different computation speeds, and the change of average entropy with respect to time was investigated.

Simulation results using different computation speeds can be summarized as follows. The average entropy vs. time graphics show that computer speed affects the stability of the system to a great extent. Especially in real time systems, when changes happen at run time, a decrease in computation speed can make the computation results useless. In our experiments we observed that in many cases, due to the increased computation times, the CDW calculations were already invalid. In other words, it was already "too late". However, since our system was a self–controlling system, it did not request the same computation load blindly at all times, the stability of the process was achieved even as the load of the computer decreased. Nevertheless, with very high computation times the system was never able to stabilize, as expected.

One of the future research directions in this area may be adding another control loop to control the number of "too late" computations. The system may try to adjust the workload of the computer by decreasing the CDW computation requests of the controllers. When the number of "too late" computations increases, some of the controllers may sacrifice by not sending any requests giving a chance to other controllers' requests to get calculated on time and in this way decrease the system entropy. Studies in this area will support the research on real-time computations.

While the speed of computation is an important factor in keeping the system stable, one needs to analyze the impact of time complexity of particular algorithms. In particular, a self-controlling system might need to decide whether to execute a more complex algorithm and possibly get better results, or use a simplistic algorithm that is only sub-optimal in terms of the system performance metric.


### Acknowledgements

We would like to thank Yong Xun and Kenneth Baclawski for their helpful suggestions and feedback.


## 7. References


[1] M. M. Kokar, K. Baclawaski, and Y. Eracar, "Control theory-based foundations of self-controlling software", *IEEE Intelligent Systems*, (May/June) (1999), pp. 37-45

[2] R. Laddaga, "Creating Robust Software through Self Adaptation", *IEEE Intelligent Systems*, (May-June) 1999, pp.26-29

[3] J. Kephard, and D. Chess, "The Vision of Autonomic Computing", *Computer*, 36(1) 2003, pp.41-52

[4] Y. Xun, M. M. Kokar, and K. Baclawski, "Control based sensor management for a multiple radar monitoring scenario", *Information Fusion: An International Journal on Multi-Sensor, Multi-Source Information Fusion*, 5 2004, pp.49-63

[5] W.E. Walsh, G. Tesauro, J. Kephard, and R,Das, "Utility functions in autonomic systems", *Proceedings of the International Conference on Autonomic Computing*, 2004

[6] M.R. Garey, and D.S. Johnson, *A Guide to the Theory of NP-Completeness*, W.H. Preeman Company, 1979

[7] J.P. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm-exact characterization and average case behavior", *IEEE Real-Time Systems Symposium*, 1989, pp.166-171

[8] http://ptolemy.eecs.berkeley.edu/ptolemyII, last accessed January 14, 2005

[9] J. Liu, J. Eker, J. W. Janneck, X. Liu, and E. A. Lee, "Actor-Oriented Control System Design: A Responsible Framework Perspective", *IEEE Transactions on Control Systems Technology*, Vol.12, No.2, March 2004

[10] J.Liu, X.Liu, and E.A.Lee, "Modelling Distributed Hybrid Systems in Ptolemy II", *Proceedings of the American Control Conference*, Arlington VA, 2001

[11] J.Liu, and E.A. Lee, "Timed Multitasking for Real-Time Embedded Software", *IEEE Control Systems Magazine*, February 2003