

EMPIRICAL EVALUATION OF A RUN-TIME DYNAMIC ADAPTABLE FRAMEWORK

JOÃO W. CANGUSSU, KENDRA COOPER, AND ERIC WONG

Abstract. Run-time dynamic adaptation is becoming a required feature for many different types of applications due to the overall complexity and distinction of environments where these applications run. SMART (State Model Adaptive Run Time), a new framework for the design of these adaptive systems, is presented here. SMART is based on the mathematics of control theory and system identification techniques allowing accurate predictions of constraint violations, such as memory overflow. Once a constraint violation is predicted, a functionally equivalent component that better copes with the resource under stress is selected. A swap to the selected component is done prior to the actual violation consequently increasing system robustness. Two examples of the application of SMART are presented to show the need and applicability of the framework. Experimental results with a prototype of SMART demonstrate the framework has an 86% accuracy in predicting and averting memory constraint violations for the first case study and a 95% accuracy for the second one. These results indicate the SMART Framework is feasible and has the potential to be a useful design solution for dynamic adaptable systems. Implementation issues such as the overhead in the prediction and run-time data collection are analyzed and improvements to the framework are proposed as future work.

Keywords: Run time adaptable systems, control theory, system identification, component-based architecture.

1. Introduction

In the past, software applications were designed to be executed on specific platforms/environments. As these environments were known a-priori, alternative solutions were hardwired to deal with the specifics of each environment. Nowadays, applications are expected to run in many distinct environments. In addition, some of these environments may not be known in advance. A question arises from this scenario: “How can we design software to cope with such dynamic environments?” Recently, dynamic adaptive systems have been proposed as a solution to

this problem. A dynamic adaptive system is one in which the system adapts itself at run-time according to changes in the environment [26].

When designing a dynamic adaptive system, new issues arise. For example, some of the inputs may not be observable such as the amount of unused memory available, the behavior is non-deterministic, the monitoring and control of the system occur continuously, and the adaptation of the system is at run-time rather than at compile-time [35].

Resource allocation has been the major technique used for dynamic adaptation. It has been applied to systems such as an operating system scheduling processes [39], QoS guarantee [30], and fault tolerance [24]. However, there is no room for reallocation when resources are not available. In this case, if an alternative functionally equivalent component that reduces the need for the scarce resource is available, a swap to this component would dynamically adapt the system to this new scenario and avoid potential failures. Therefore, adaptation can be based not only on resource allocation but also on the swap to alternative design choices that cope better with resource constraints. The expectation that the same application can be executed on very distinct platforms exemplifies this need. For example, web applications are now having to handle the problem of being used not only on personal computers, but also on cell-phones and personal digital assistants [14]. The capacity of these platforms range drastically from very powerful workstations to low memory, low performance pocket devices. The {re}allocation of resources, performed by the operating system, does not appear to be a solution under such distinct scenarios. For example, a real-time operating system cannot do anything if an application requires 10MB of memory and the total available memory on the system is 5MB. An adaptive application should be able to identify such environmental constraints and swap to an alternative, though functionally equivalent, component that can be executed under the current conditions. The same adaptation approach can be used when run-time bottlenecks are predicted or detected. In addition, robustness can be improved if some properties of the alternative choices can be guaranteed, such as the maximum memory required under any scenario.

In this work we propose a new approach to the design of adaptive systems called the SMART (State Model Adaptive Run Time) Framework. SMART innovatively integrates two areas of leading edge research in software engineering. The first is the use of feedback control theory to support the continuous monitoring and control of the system in a dynamically changing environment [9]. The second area is the effective specification, matching, and selection of off-the-shelf (OTS) components. SMART provides a knowledge-based repository of components. The functional and non-functional behavior of the components are rigorously specified in the extensible markup language (XML); the system can run XML queries that identify a component (or set of components) that better suits the environmental changes predicted by a controller. Once identified, the components are swapped into the system.

This paper is organized as follows. An overview of control theory background is presented in Section 2. The objectives of the SMART Framework are presented in Section 3.1; the framework is described in Section 3.2. Two examples are used to illustrate and empirically evaluate the SMART Framework in Section 3.3. Implementation issues associated with the examples are presented in Section 3.4. After the presentation of related work in Section 4, the conclusions we have drawn regarding the SMART Framework and future work are delineated in Section 5.

2. Control Theory Background

Linear state feedback models have provided useful representations for large classes of engineering, biological, and social processes [11, 19]. In this section we present concepts and definitions of state models and system identification techniques that are the most relevant to our approach.

The general format of a LTI (Linear Time Invariant) state model is presented below:

$$(1) \quad \begin{cases} \dot{x}(t) = Ax(t) + Bu(t) \\ y(t) = Cx(t) + Du(t) \end{cases}$$

where $x(t)$ is the state vector representing the dominant variables characterizing the process; $u(t)$ is the input signal; $y(t)$ is the output/measurable variables; and A , B , C , and D are the coefficient matrices.

A model capturing the behavior of a system must be available if such system is to be controlled. A model that can capture the dominant behavior of different aspects of any adaptive system is unlikely to be statically created due to the variety of scenarios and environments where these systems execute. However, based on the observation of the behavior of a system executing in a specific environment a model can be dynamically customized for this scenario. Ljung states "System identification deals with the problem of building mathematical models of dynamical systems based on observed data from the system." [29]

An adaptive system can be designed in a such a way that any or some specific inputs/outputs are shared with other applications at the same time environment resources are monitored. Under this scenario, the ingredients to apply system identification techniques are present and a model can be inferred from the observations. A number of techniques, such as least-square and markov parameters [29], are available to identify state space models. In order to implement a fast prototype of SMART, the least-square identification procedure available in MATLAB was chosen; a concise description of this technique is presented below.

Let A , B , C , and D be the matrices in Eq. 1 as described earlier and organize them as indicated below.

$$(2) \quad Y(t) = \begin{bmatrix} x(t+1) \\ y(t) \end{bmatrix} \Theta = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \Phi(t) = \begin{bmatrix} x(t) \\ u(t) \end{bmatrix}$$

Using the definitions of $Y(t)$, Θ , and $\Phi(t)$ above, we can rewrite Eq. 1 as $Y(t) = \Theta\Phi(t)$. The observation of the system resources and the inputs/outputs of the application are used to construct the vectors $Y(t)$ and $\Phi(t)$. Then, a least square approach as in Eq. 3 can be used to compute an approximation ($\hat{\Theta}_N^{LS}$) for the matrix (Θ).

$$(3) \quad \hat{\Theta}_N^{LS} = \left[\frac{1}{N} \sum_{t=1}^N \Phi(t)\Phi^T(t) \right]^{-1} \frac{1}{N} \sum_{t=1}^N \Phi(t)Y^T(t)$$

The matrix $\hat{\Theta}_N^{LS}$ represents the approximation for the coefficient matrices A , B , C , and D in Eq. 1. They form the state model capturing the dynamics of the system under consideration. The state model is then used to predict and control the behavior of the system.

3. The SMART Framework

The SMART Framework is based on the use of control theory to manage the adaptation issues when resource constraints and multiple design choices are available. A linear approximation of the system resources and the application is expected to capture their dominant behavior. This approximation has been proven to be general enough for such representation [30]. After a model is specified, feedback control can be applied to predict and regulate (select alternative design choices) the behavior of the system.

3.1. Objectives

The overall goal of the SMART Framework is to provide an environment for the design of dynamic adaptive software which allows the use of multiple solutions based on system resources and constraints. The components used to build the alternate solutions may be available when the system is launched or included while the system is running. In either case, the swap from one set of components to another is done at run time. The assumption here is that the selection of a set of components is made that better uses the available system resources of interest (e.g., CPU, memory, bandwidth, etc) and therefore improves the overall quality of the system.

To accomplish this goal, the SMART Framework integrates two areas of leading edge research in software engineering. The first is the use of feedback control theory to support the continuous monitoring and control of the system in a dynamically changing environment. Although feedback control theory is well-known in a variety of engineering areas, its application to software engineering is relatively new [7, 11, 27]. The second area is the effective specification, matching,

and selection of off-the-shelf (OTS) components. SMART provides a knowledge-based repository of components. The functional and non-functional behavior of the components are rigorously specified; the system can run queries that identify components that better utilize a resource. Once identified the components can be swapped into the system to better meet the needs of the users. If these components are not yet integrated/loaded into the system, then the SMART Framework needs to make use of dynamic programming languages such as Java. The integration of these two areas of research are reflected in the architecture of the SMART Framework.

3.2. Architecture

As shown in Figure 3.1, the architecture of the SMART Framework is composed of the design option repository, actuator, pre-replacement, current component, system identification module, resource monitor, and controller. The architectural components and their relationships are described below.

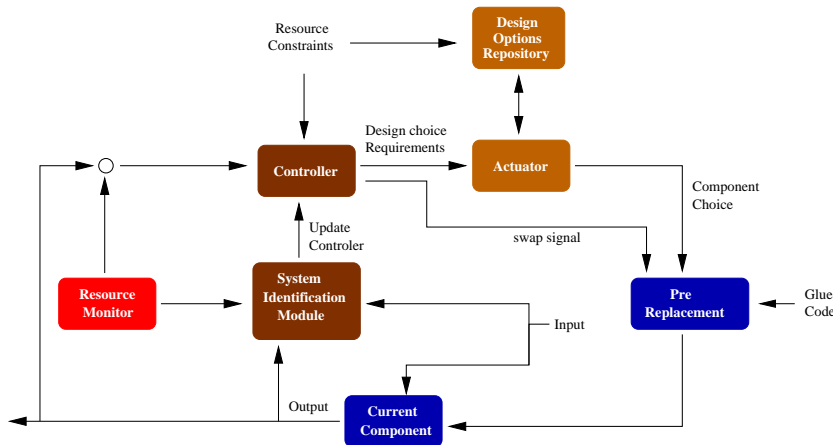


Figure 3.1: Overview of the SMART Framework architecture.

Design Option Repository: This repository contains the components which are available to provide alternative design solutions. For example, if a system needs to multiply large sparse matrices, then two component choices may be available:

(i) a fast but high memory usage approach, and (ii) a slow but low memory usage approach.

Each component is formally specified to describe its functional properties, non-functional properties (e.g., CPU usage, memory usage, interface requirements, etc.), its composition (a component may be composed of one or more components), and the location of the code in the system. When the actuator queries the repository, the algorithm to determine the best possible matches also uses the resource constraints specified for the system.

The formal notation used to define the components is the Extensible Markup Language (XML) [3]. The advantages of using this notation include that it is straightforward to learn and that tool support such as parsers, editors, and browsers are available.

Each alternative solution (i.e., a collection of one or more components) must be smoothly incorporated to the system and therefore each component should be designed such that it can be integrated with the whole system without recompilation and relinking. Techniques such as structural conformance, delegation, and wrapping may be used to achieve this goal [18].

System Identification Module: In order to use control theory [19] one needs a mathematical model (i.e., a system of equations) that relates the inputs and outputs of the system. Since the components and their interactions affect the systems resources and, as a result, a specific model is unlikely to accurately capture the dominant behavior of all the components. However, system identification techniques [29] allow for the dynamic determination of tailored models for a specific component/system.

This part of the SMART Framework is similar to the work done for autonomous agents [28]. By observing the output from the Software Product (component) and the systems resources provided by the **Resource Monitor**, the **System Identification Module** relates them to the input and creates a state model (**Controller**) that captures the dominant aspects of this relationship. The precision of the model increases as more data (input, resources, and output) become available and the **Controller** is updated.

Controller: The controller predicts when a bottleneck constraint is expected to occur and determines the latest possible binding time to correct the problem. This prediction allows for the minimization of the overhead costs associated with component swap. The controller uses the *resource constraints* and the current state of the system to make this prediction.

Resource Monitor: The resource monitor represents an auxiliary tool that monitors the resources of the environment and sends this information to the **Controller** and the **System Identification Module**.

Current Component: This block represents the current design choice that is running. It receives the input and passes information (output) to the **Controller** and the **System Identification Module**.

Actuator: The actuator selects one of the design options provided by the design option repository. The selection algorithm uses the desired system requirements from the controller that should be met to achieve the system's integrity, performance, and consistency. The notation used to query the **Design Component Repository** is XML SQL [32].

Pre-Replacement: This pre-replacement module prepares the component selected by the actuator to replace the current one. It inserts any necessary glue code and links it to the component. The **Pre-Replacement** then waits for a signal from the controller determining the exactly time when the swap should occur. This technique improves the performance by preparing the component in advance.

A prototype of SMART has been implemented using MATLAB scripts for the system identification module and the controller. The design option repository has been specified in XML. The resource monitor and the actuator are implemented in C.

3.3. Application and Evaluation

Two case studies are presented next. In both cases a trade-off between execution time and memory usage can be clearly identified. The case studies demonstrate that the need for an adaptive framework like SMART exists whenever there are constraints associated with design choices.

3.3.1. Case Study I: Matrix Multiplication

In this example, a system needs to perform a sequence of square matrix multiplications; the matrices are highly sparse (i.e., 5%). There are two components available (Figure 3.1: Design Options Repository) to perform the multiplication of the matrices:

- C_1 : this component is designed to have high performance (i.e., it is fast) but demands high memory usage. It is implemented by allocating a two dimensional array for each matrix.
- C_2 : this component is designed for low usage of memory and therefore does not have good performance under certain scenarios. It is implemented by allocating a one dimensional array of pointers for each matrix; these pointers reference a linked list of non-zero elements for each row in the matrix.

A trade-off between memory usage and performance can be inferred from the description of the components C_1 and C_2 .

A sequence of experiments has been conducted as a first step in the validation of the SMART Framework. Scenarios with a variety of sequential and concurrent launches of components C_1 and C_2 have been analyzed [8]. Due to space constraints, only the results of the scenarios described below are presented here. These scenarios are designed to measure the memory, CPU time, accuracy of the framework, and the impact, if any, the implementation language (C++, Java) has on the behavior of the components.

The expectation is that the use of C++ makes the trade-off between memory use and performance more distinct while still allowing runtime adaptation. That

is, the loaded components can be used for runtime adaptation but the insertion of a new component requires recompiling and linking the application.

The alternative use of Java as the programming language makes it possible to insert a new component at runtime through its dynamic loading properties. However, a quantitative analysis of the impact of the Java garbage collector on the accuracy of the framework is defer to future work. The expectation is that some delay in releasing the unused memory results as memory usage increases for component C_2 .

Experimental Scenario I (C++)

In the first scenario, the components are implemented in C++ and are executed sequentially. The example system, launched as a single process, initially executes component C_1 by default (Figure 3.1: Current Component) . The system performs 20 iterations of the following calculations: the system executes a sequence of square matrix multiplications of order n , where n ranges from $n = 400, 430, 460, \dots, 670, 700$; this sequence of calculations is repeated three times.

The use of system identification techniques (Figure 3.1: System Identification Module) in the framework to produce a linear state space model (Figure 3.1: Controller) for changing environments allows the prediction of the future behavior of the application. A horizon of seven steps ahead is used to predict the behavior of the components. The constraint on the system that needs to be monitored and controlled is the memory utilization. Here, the constraint is that the memory use must not exceed a specified threshold of memory usage (Figure 3.1: Resource Constraints).

As the system is executing, the framework's prediction capabilities are used to make the swap to execute component C_2 . The swap (Figure 3.1: Actuator + Pre-Replacement) from one component to another may occur at any time. If a prediction that the threshold constraint will be violated, the current multiplications are aborted for component C_1 and start over using component C_2 . It should be clear that robustness and not performance is the major goal for this scenario.

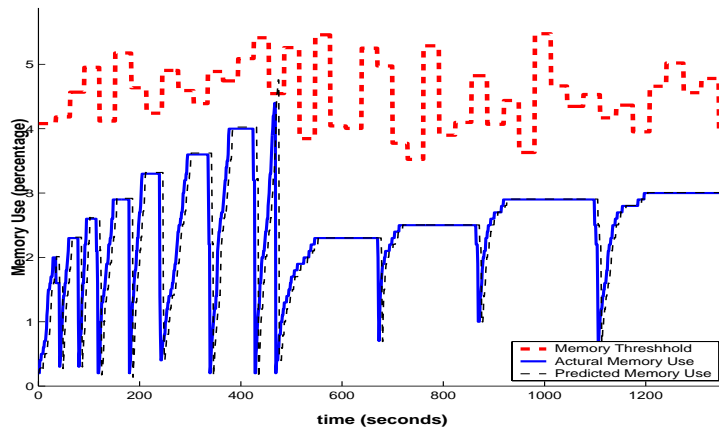


Figure 3.2: Scenario I: Memory usage vs. CPU time for C++ components

Additionally, to make the experiment more realistic, the memory threshold changes randomly at runtime ranging from 3.5% to 5.5% of overall memory use. This changing threshold emulates an environment in which processes are being continuously created and terminated.

Figure 3.2 show the results for the first scenario where it can be seen that the threshold is changing dynamically (this is the heavy-dashed line near the top of the Figure). The framework's predictions of memory usage are in the light-dashed line; the actual memory used is in the solid line. Memory usage is collected by the Resource Monitor from Figure 3.1.

An interesting result demonstrating the prediction of a memory constraint violation, swapping from component C_1 to C_2 (which uses less memory), and the resulting aversion of the constraint violation near the time $t = 470$ seconds is shown in Figure 3.2. We zoom in on this region of time and illustrate it in Figure 3.3.

At time $t = 470$, the actual memory use is 4.3%; the prediction made is that the system is going to (in seven steps) violate the constraint of crossing the threshold of 4.5% of memory usage. At this point, C_1 is swapped out for component C_2 . As a result, the threshold is not exceeded and the system complies with its constraint. An increase in the robustness of the application can be inferred from these results.

As the 20 iterations of the matrix multiplication sequence are executed, an average of 9 swaps per iteration between components C_1 and C_2 is measured. The

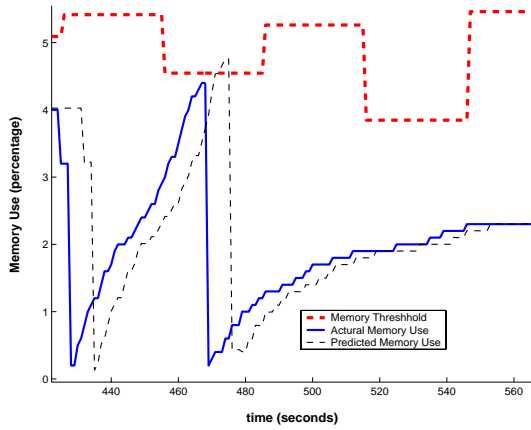


Figure 3.3: Scenario I: time window demonstrating prediction, swapping, and aversion of memory constraint violation

swap to an alternative component has prevented memory overflow according to the threshold constraint in 86% of the cases. These results are discussed in Section 3.4. The 86% success rate is a good indication of the potential of the framework proposed here.

Experimental Scenario II (Java)

The purpose of this scenario is to measure memory usage and CPU time for the components and allow a comparison of the behavior of these components, written in Java, with the components written in C++.

In the second scenario, the components C_1 and C_2 are implemented in Java and are sequentially executed. The following calculations are performed: the system executes a sequence of square matrix multiplications of order n , where n ranges from $n = 300, 330, 360, \dots, 580, 610$; this sequence of calculations is repeated four times. The range is reduced in comparison to the previous scenario, in anticipation of a large amount of CPU time expected to execute the matrix multiplications.

Figure 3.4(a) presents the performance results with respect to the order of the matrices. According to the collected data, the performance of the JAVA components demonstrate the Java version of C_1 uses approximately 16 seconds to

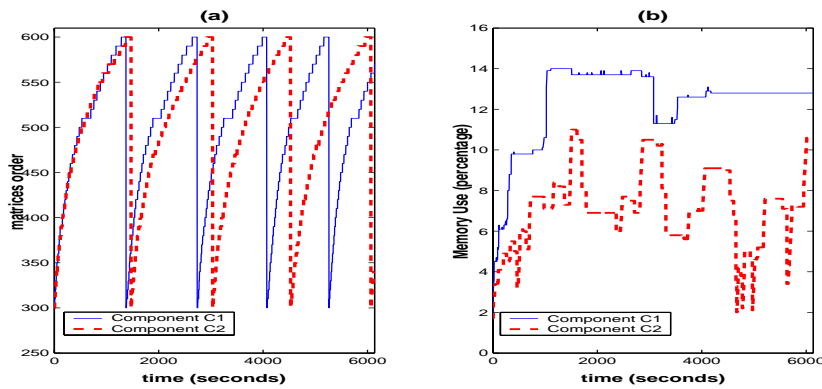


Figure 3.4: Scenario II. Matrices order, memory use percentage vs. CPU time (Java)

calculate a matrix multiplication of order 400, whereas the C++ version uses less than 2 seconds. The slower performance of the Java components in comparison to the C++ components is an expected result. Also, in another scenario [8] where the Java versions of C_1 and C_2 were executed concurrently, C_1 was five times faster than C_2 but was only 10% better when the two components were launched sequentially as Scenario II. As expected, the behavior of equivalent components implemented using different programming languages impacts the performance and consequently the control of an application.

If we assume a deterministic behavior for the scenarios described above, then their observed results could lead to the specification of a static controller for the two components. However, such controller would, most likely, not be useful when considering other scenarios such as the simultaneous execution of $N = 3, \dots, 10$ processes using one of the two design choices. Another situation to consider would be the addition of a third design choice. In any of these cases the controller would need to be tailored to fit the distinct scenarios. A dynamic self-tuning mechanism for the controller would avoid its re-definition in both cases.

Furthermore, without a complete knowledge of all the elements executing in the system at the same time, we cannot make the assumption of a deterministic behavior for the components and therefore, a scenario may produce different results

at different times. Again, a run-time self-tuning mechanism for the controller appears to be the solution to handle such dynamically changing environment.

The memory use of the Java components yielded interesting results. In Figure 3.4(b), the memory use associated with C_2 oscillates as the execution time progresses. This corresponds to changes in the order of the matrices. In contrast, the memory use tends to stabilize at a certain level when it is associated with component C_1 .

In addition, when the components are implemented in Java instead of C++, the memory used by C_1 does not drop even when repeating the cycle by re-starting with matrices of order 300. The Java garbage-collector, even when explicitly invoked, does not free all the unused memory for component C_1 . The same is not true for component C_2 where the garbage collector is more effective. The measurements indicate that the dynamic allocation of memory is the reason for such behavior.

The examples presented in this case study are useful in identifying some of the issues to be addressed by the SMART Framework. It also serves to show the applicability of the framework. More experiments are being designed and run to provide additional evidence of the applicability of the SMART framework. Recently, a data persistence case study has been investigated. A summary of this example is provided below.

3.3.2. Case Study II: Data Persistence

The SMART Framework has also been recently applied to an example system that uses a data persistency service. The selection of a persistency service is a decision that may affect the performance, memory requirements, data integrity, and other non-functional behaviors of a system. Persistent data are, simply put, data that exist after a program executes. Commonly, the data are saved in a file or a database on disk. The data persistency service provides a mechanism to save the data; there are numerous algorithms available. The algorithms differ in response time performance, data integrity, and the amount of temporary data storage (TDS) memory needed. The algorithms are presented in the context of a data service architecture [20]. The sequence of events to retrieve an entity into the TDS begins

with a client request. Here, a client is another module or subsystem that needs to use the data service. If the entity is in the TDS, then it is returned to the client. Otherwise, a request is made to retrieve the entity from the persistent data store and store it in the TDS. The entity is located and retrieved, likely in an optimized form (e.g., compressed, encrypted, fragmented, etc.) and converted into a form suitable for the TDS (e.g., uncompressed, decrypted, defragmented, etc.) and ultimately, the requesting client. When the entity needs to be saved to disk, the converters transform the TDS entity into the optimized form and determine where to save it. Eventually, the TDS entity is removed.

We present two options that are used to represent some interesting non-functional trade-offs for memory, response time performance, and data integrity. We recognize that there are many possible data service algorithms, for example, those that utilize established caching algorithms such as least recently used, least frequently used, etc. Additional options are going to be investigated in future work.

The first option (component P_1) uses an algorithm which involves saving an entity immediately after modifying it, like a write-through caching algorithm. However, once the entity is saved, it is not kept in the TDS. When a client request is made for an entity, e_i , the TDS requests the entity from the Persistency Data Service (PDS), e_{iPDS} . The entity is located, retrieved, converted, and the TDS is updated to include the entity, e_{iTDS} . After the entity e_{iTDS} has been modified, then it is converted into e_{iPDS} , the location is determined, and the entity e_{iPDS} is stored. When the application terminates, the resources for entity e_{iTDS} are released. In summary, this option provides a solution that can be characterized as low memory, slow response time performance, and high data integrity.

A second option (component P_2) uses an algorithm that saves all modified entities in memory to disk periodically; the entities remain in memory after being saved. When a client request is made for an entity, e_i , the entity may or may not be in the TDS. If it is, then the entity is simply returned to the client. Otherwise, the TDS requests the entity (e_{iPDS}) from the PDS. The entity is located, retrieved, and converted. The TDS is updated to include the entity, e_{iTDS} , and it is returned to the client. The entity e_{iTDS} may be accessed or updated multiple times without being saved to the PDS. Here, a periodic timer determines when modified entities

in the TDS are converted, locations determined, and stored in the PDS. When the application terminates, if entity e_{iTDS} has been modified, then it is converted, location determined, and stored as e_{iPDS} . Subsequently, the resources for each entity e_{iTDS} are released. In summary, this option provides a solution that can be characterized as high memory, fast response time performance, and moderate data integrity.

Each solution is evaluated by running the component and gathering data about the solution (e.g., how much memory it uses, how much time is used, etc.) for a specific environment.

An application, which uses a data persistency component, adapts to the changing amount of memory in the environment. When the amount of available memory is low, component P_1 is a better choice, while P_2 is a better choice when more memory is available. Furthermore, let us assume that the application has a memory constraint that limits the use of component P_2 . Let us arbitrarily define a 7% memory threshold that if crossed would cause the crash of the application. Under these conditions, the two possible swaps are:

- Swap from P_2 to P_1 : occurs once a memory usage of 7% or more is **pre-dicted** by the SMART Framework.
- Swap from P_1 to P_2 : occurs when more than 5% of memory is available in the system.

The persistency service application have been executed over a period of 8.601 hours with a total of 1812 swaps (906 from P_2 to P_1 and 906 from P_1 to P_2). Out of 906 swaps from P_2 to P_1 to avoid violating the constraint, only 37 were not executed in time, i.e., the memory threshold was crossed in 4.08% of the swaps from P_2 to P_1 (refer to Section 3.4 for details). This would hypothetically crash the persistency service application, but for experimental purposes, only the number of failed swaps are recorded.

3.4. Implementation Issues

In the examples presented in Sections 3.3.1 and 3.3.2, memory overflow with respect to a predefined threshold occurs in some cases. However, in almost all

such cases memory violations are not due to wrong predictions but due to late reaction time. That is, SMART has predicted the constraint violation before it happened but due to a delay in the swap, the violation occurred. Possible reasons for the delay include (a) the overhead of running the MATLAB environment and the monitoring process, (b) the overhead in the communication between the MATLAB scripts (for the system identification and the controller) and C threads (for the actuator and the resource monitor), and (c) the scheduling of the “Actuator” thread.

The scheduling problem and the communication overhead can be minimized by the use of inter-process communication. The overhead in the prediction could be also minimized by having a compiled version of the MATLAB scripts. However, not all functions in the scripts can be converted to C code, mainly the methods in the System Identification Toolbox. A partial solution to decrease the overhead is the decomposition and conversion of only parts of the scripts to C code. Another alternative to decrease the number of violations is to increase the number of steps ahead used in the prediction. This would give more reaction time to the application. On the other hand, the accuracy of the predictions decreases as the number of steps ahead increases and a better synchronization algorithm is needed to avoid premature swaps.

We have noticed that the task of reducing the overhead to a minimal level can lead to a potential limitation in the use of the SMART Framework. When a prediction needs to be made for a very short time ahead, say less than ξ milliseconds, the overhead in the framework may not allow the swap between the current component and the component that should be used to be completed in such a short time. As a result, some system constraints may be violated. Though better implementation and design choices can reduce the overhead, it is still premature to determine a value for ξ .

The memory required by SMART could be a problem for low memory platforms such as many embedded applications. A compiled version of the MATLAB scripts can drastically reduce the memory overhead to a level that becomes almost insignificant when compared to the memory required by the application. However,

this is restricted by the capability of compiling the system identification functions in future releases of MATLAB.

There is also another challenge in using SMART. The techniques for identifying state space models are available and have been successfully applied [10, 30] to the calibration of many different types of systems under distinct scenarios. However, it is not clear whether the use of one specific technique, for example, the least square approach, can produce reasonable results for different applications and systems. As a result, it might be necessary to apply the SMART Framework recursively to select appropriate calibration techniques/procedures for different scenarios based on the available data and system/resource constraints.

4. Related Work

The SMART Framework is an adaptive approach based on the selection of components available in a repository. Therefore, the work most related to SMART, adaptive systems and component-based architecture, is described next.

4.1. Adaptive Systems

Self-adaptive systems has been proposed for specific domains such as QoS [39, 30], ad-hoc network management, traffic control, and signal processing [8]. The world wide web has also been the focus of the use of adaptive techniques at “interface” level as well as at a lower level [8]. Approaches that are not domain specific have also been proposed [14]. Many of successful approaches are based on the sound theory of feedback control [8, 30, 39]. Decision theory is another technique used in adaptive systems. Uncertainty has common place when dealing with adaptive systems; and decision theory provides mathematical tools, such as game theory and subjective probability, to address decision making problems under the presence of uncertainty.

A large body of methods has also resulted from the Dynamic Assembly for Systems Adaptability, Dependability, and Assurance (DASADA) sponsored by DARPA [15, 21, 41]. Though an overlap can be identified between many DASADA

projects and SMART, the predictability feature presented by SMART clearly distinguishes it. A large part of the DASADA projects focuses on self-healing techniques while SMART focuses on preventing the errors (related to non-functional requirements) from occurring in the first place. The concepts of probes, gauges, and events, among others and the corresponding techniques developed by the DASASA projects are of extremely importance for this project and are being further investigated. It is clear that the DASADA projects and SMART complement each other.

Next, a brief description of three distinct adaptive techniques is presented. They are selected as representatives, among a multitude of other approaches [8], due to their relation to the SMART Framework. *Containment Units* approach has a similar goal of designing/specifying adaptable components while the QoS framework uses a similar, control theory technique that is applied in SMART.

4.1.1. Containment Units

A *Containment Unit* is basically a module that implements some functionality. The basic difference between a *Containment Unit* and a regular module is that the first is also defined according to nonfunctional requirements, such as time, memory, and sensors [14]. The interface of a *Containment Unit* is defined by a tuple (F, R, CP, FC) . The functional requirements are represented by F and the nonfunctional by R . The expected input and output, including internal faults is represented by CP (Communication Protocol). External faults due to operational components but that are handled inside the *Containment Unit* are specified in FC [14]. The architecture of a *Containment Unit* is composed of an “Operational Component”, an “evaluator”, a “change agent”, an “Adapter”, and an “Implementation”. A *Top* component initializes the unit and manages the communication protocol. A set of operational components provides the functionality of the *Containment Unit*. The *Evaluator* monitors the performance of the operational components in order to ensure that the specification of the *Containment Unit* interface is satisfied. When the *Evaluator* determines that the unit is not performing according to the specification, the *Change Agent* is in charge of selecting a different operational

component that better copes with the current resources. Another alternative is the re-allocation of resources to match the current operational component [14].

4.1.2. Adaptive Control Framework for QoS

The work done on adaptive systems for QoS guarantees [30] is based on the application of control theory aspects with an automatic adjustment of the controller by means of system identification techniques. A pole placement adaptive control technique is used to overcome the restrictions of a static designed controller. The goal is to maintain a specified target hit ratio performance by adjusting storage allocation. As can be observed in Figure 4.5, the *Automatic Model Estimator* uses the system input and output to update the controller. The *Controller* monitors the system and signalizes the *Actuator*, when necessary, to re-allocate resources.

The SMART Framework has a similar architecture to the one presented by Lu and Abdelzaher [30]. Both approaches are based on control theory and on automatic tuning of the controller. SMART differs from the QoS framework by adjusting the system not based on resource allocation but on swapping to a component that provides the same functionality but better copes with the environment status. Also, as described in Section 5, SMART is a more generic approach by allowing the specification of the system constraints and their relation to the alternative choices.

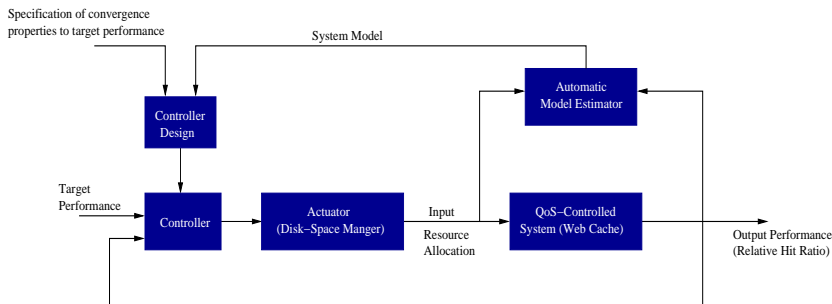


Figure 4.5: Adaptive Control architecture for Cache QoS Guarantees [30].

4.1.3. Chameleon

Though specific to fault tolerance, Chameleon [24] is a good example of an adaptive system designed for the use of COTS (Component off-the-shelf). The adaptive structure of Chameleon is based on three different types of components: managers, daemons, and common. These components control all the operations in Chameleon and are named ARMORs (Adaptive, Reconfigurable, and Mobile Objects for Reliability). Chameleon is initialized through the invocation of a Fault Tolerance Manager (FTM), which invokes two other ARMORs to handle communications and failure detection. A redundant FTM is also created to increase dependability.

Users submit applications and associated availability requirements to Chameleon. The FTM is in charge of selecting an appropriated “fault tolerance execution strategy” [24] to cope with the specified requirements. This strategy is not fixed and can be easily reconfigured. An ARMOR manager is responsible for the execution of the selected strategy guaranteeing the availability requirements.

4.2. Component-Based Architecture

The issues involved with developing a complex system with OTS components are extensive; they impact almost every aspect of software development including the requirements specification, architecture, implementation, testing, maintenance, OTS selection and evolution, project management, cost estimates, licensing, etc. Specific solutions have been proposed to address how to evaluate, specify, select, and compose OTS components. A survey of work in these areas is available in [8].

As work has progressed in the area of COTS research, specific solutions have been proposed to address some of these issues. Solutions to the problems of how to evaluate [4, 5, 17], specify [1, 33, 37], select [5, 6, 46, 34, 36], and compose [2, 42] OTS components have been proposed in the literature. Methodologies to support these activities have also been described [13, 31]. Issues related to testing [22] and maintaining [12, 40] OTS intensive systems have also been presented.

Repository based approaches, which address a collection of issues including the evaluation, specification, selection, and composition of OTS components, have also been proposed [38, 43, 44, 45]. As our research is most closely related to work focused on the design of component based systems using a repository, we present some of the proposed solutions in this area.

4.2.1. On-line Repository for Embedded Systems

A component-based approach to embedded software development called ORES (Online Repository for Embedded Software) [43] provides a repository of components and tools for component specification, composition, and analysis at design time. In this semi-automated approach, the developer interacts with the tools to create a component-based system. The ORES approach uses ontology based repository browsing. The components are stored in the repository using a hierarchy of domain, sub-domains, and packages. A document type definition (DTD) is used for information attributes definition and the extended markup language (XML) is used to specify the actual measurement data for each component. Due to the hierarchical nature of the repository, a child node inherits the attributes from the parent and we can add or delete attributes to form its own set. The node information contains general information (identification, type, keyword list with weight, short description of the node), information pointers (pointers to file names for modules of code or documentation), and properties (reliability, portability, resource requirements, etc.)

4.2.2. K-BACEE

The Knowledge-Based Automated Component Ensemble Evaluation (K-BACEE) approach [38] is a partially automated, knowledge-based approach to evaluating ensembles of components within the context of a system requirements specification (SRS). K-BACEE emphasizes the identification and selection of ensembles, or groups, of components to satisfy the requirements of a system. The main activities include the iterative development of an SRS, identifying individual components that meet the SRS using queries, identifying ensembles of components that

are compatible using integration rules, defining functional and non-functional attributes of COTS components, and defining the integration rules (i.e., rules that determine if components are compatible). The SRS defines the functional and non-functional requirements of the system. The attributes of the COTS components define its functional behavior and the non-functional characteristics that impact compatibility with other components, for example, the protocols supported or the implementation language of the component. The integration rules define how the values of component attributes affect their integration. The SRS, components, queries, and integration rules are stored in the knowledge base. A Java prototype of K-BACEE tool support has been developed, the requirements specification and components are defined in the extensible markup language (XML); queries are defined in the extensible query language (XQL). The integration rules are defined using an object-oriented rule-based programming language, JRules. Once the functionality and constraints of the system are defined, the SRS is manually converted into a series of queries on the component repository. Components that match the requirements specified in the SRS are grouped into ensembles by the tool support; their compatibility is evaluated based on the value of attributes and a repository of software engineering integration rules. The integration rules are executed to rank the ensembles; the ensembles are presented to the K-BACEE user for selection.

A general overview of the comparison of the adaptive and component based techniques is shown in Table 1. As can be observed, SMART has the strengths of both worlds, in particular the ability to predict, and therefore avoid, constraint violations.

5. Conclusions and Future Work

The objectives, description, and advantages of a new adaptive framework named SMART have been presented here. The overall goal of this framework is to provide an environment for the design of adaptive software which allows the dynamic use of multiple solutions based on system resources, component constraints, and user requirements.

Table 1: Comparison of Adaptive and Component Based Techniques

Techniques → ----- Features ↓	S M A R T	Q o S	C o n t. U n i t s	C h a m e l e o n	O R E S	K - B A C E E
Flexibility	√		√			√
Resource Allocation Based		√	√			
Component Swap Based	√		√			
Component Oriented	√		√	√	√	√
Dynamic Adaptation	√	√	√	√		
Predictability	√					
Repository Based	√		√		√	√
Specialized Domain		√		√	√	

Significant advantages of the SMART Framework include that it is domain independent and it supports the prediction of future behavior, which supports swapping components very efficiently. Based on its capabilities, the SMART Framework appears to be a more complete alternative for the designers of adaptable systems. SMART impacts not only on the way adaptive systems are conceived but also on the mechanisms used to control them. In addition to the advantages of many adaptive system approaches (self-adaptation, robustness, better resource usage, etc.), the SMART Framework provides two distinguishing features.

- **Flexibility:** The majority of the adaptive techniques are domain specific. This is due to the fact that partial or complete knowledge about the relationship of the inputs and systems resources have to be known a-priori. The

System Identification Module in the SMART Framework dynamically identify these relationships and therefore allows its application to a variety of different domains.

- **Predictability:** The availability of a state model capturing the dominant behavior of the system allows the prediction of future behavior. The time when changes are necessary can be predicted and the system preparation to swap to a new component can be done in advance resulting in an overall increase in the systems performance and robustness.

In the experimental evaluation, SMART successfully predicted and prevented memory overflow in 86% of the cases for the first case study and 95% for the second. This is a good indication of the viability and accuracy of the framework. Possible techniques that may be used to improve the accuracy of the framework include creating a compiled version of the MATLAB scripts in order to reduce processing delays. This feature is supported in the MATLAB tool and is going to be investigated in the next step of the work.

Additional verification of the SMART Framework is needed. In the next step of the work, SMART is going to be evaluated to determine how effective it is for applications that are distributed over heterogeneous platforms such as web systems. Experiments including the monitoring and control of multiple, concurrent processes for a variety of example systems are also planned.

References

- [1] E. Addy and M. Sitaraman. Bridging the gap between cots product reuse and formal methods: A case study. Technical Report NASA-IVV-98-016, NASA/WVU Software Research Lab, Fairmont, WV, 1998.
- [2] K. Balasubramanian, N. Wang, C. Gill, and D.C. Schmidt. Towards composable distributed real-time and embedded software. In *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 226–233, 2003.
- [3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0*. W3C Recommendation 6, second edition edition, October 2000.
- [4] E. Brenner and I. Derado. Specifying a certification process for cots software components using uml. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 291–297, 2001.
- [5] L. C. Briand. Cots evaluation and selection. In *Proceedings of the International Conference on Software Maintenance*, pages 222–223, 1998.
- [6] X. Burgues, C. Estay, X. Franch, J. Pastor, and C. Quer. Combined selection of cots components. *Lecture Notes in Computer Science* 2255, 2002.
- [7] João W. Cangussu. A software test process stochastic control model based on cmm characterization. *Wiley Interscience - Software Process: Improvement and Practice*, (2):55–66, April/June 2004.
- [8] João W. Cangussu and Kendra Cooper. A new approach for the design and control of adaptive systems. Technical Report UTDCS-21-03, University of Texas at Dallas, Richardson-TX, USA, May 2003.
- [9] João W. Cangussu, Kendra Cooper, and Changcheng Li. A control theory based framework for dynamic adaptable systems. In *19th Annual ACM Symposium on Applied Computing, SAC 2004*, Nicosia, Cyprus, March 14 -17 2004. ACM.
- [10] João W. Cangussu, Raymond A. DeCarlo, and Aditya P. Mathur. A state model for the software test process with automated parameter identification. In *Proceedings of the 2001 IEEE Systems, Man, and Cybernetics Conference (SMC 2001)*, pages 706–711, Tucson, Arizona, October 2001.
- [11] João W. Cangussu, Raymond A. DeCarlo, and Aditya P. Mathur. A formal model for the software test process. *IEEE Transactions on Software Engineering*, 28(8):782–796, August 2002.
- [12] D. Carney, S. A. Hissam, and D. Plakosh. Complex cots-based software systems: practical steps for their maintenance. *Journal of Software Maintenance: Research and Practice*, 12(6):357–376, Nov/Dec 2000.
- [13] Lawrence Chung and Kendra Cooper. Defining goals in a cots-aware requirements engineering approach. *Systems Engineering journal*, 7(1):61–83, 2004.

- [14] Jamieson M. Cobleigh, Leon J. Osterweil, Alexander Wise, and Barbara Staudt Lerner. Containment units: A hierarchically composable architecture for adaptive systems. In *Proceedings of the 10th International Symposium on the Foundations of Software Engineering (FSE 10)*, pages 159–165, Charleston, SC, November 2002.
- [15] C. Dabrowski and K. Mills. Understanding self-healing in service-discovery systems. In *Proceedings of 1st ACM Sigsoft Workshop on Sel-healing Systems (WOSS'02)*, pages 15–20, Charleston, South Carolina, November 18-19 2002. ACM Press.
- [16] Samir Dami, Jacky Estublier, and Mahfoud Amiou. *Process Technology*, chapter APEL: a Graphical Yet Executable Formalism for Process Modeling. Kluwer Academic Publishers, January 1998. Also published as a Special Issue of Journal of Automated Software Engineering.
- [17] E. Dubois and X. Franch. Models and processes for the evaluation of cots components. In *Proceedings of the 26th International Conference on Software Engineering*, pages 759–760, 2004.
- [18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [19] Graham C. Goodwin, Stefan F. Graebe, and Mario E. Salgado. *Control system design*. Prentice Hall, Upper Saddle River, New Jersey, 2001.
- [20] G. Govi. Data service documentation. <http://pool.cern.ch/storage/doc2/DataSvc.pdf>, 2003. Version 0.5.
- [21] Philip N. Gross, Suhit Gupta, Gail E. Kaiser, Gaurav S. Kc, and Janak J. Parekh. An active events model for systems monitoring. In *Working Conference on Complex and Dynamic Systems Architecture (CDSA)*, Brisbane, Australia, December 12-14 2001.
- [22] S. Hissam and D. Carney. Isolating faults in complex cots-based systems. *Journal of Software Maintenance: Research and Practice*, 11(3):183–199, 1998.
- [23] X. Illa, Franch X, and J. Pastor. Formalising erp selection criteria. In *Proceedings of the 10th International Workshop on Software Specification and Design*, pages 115–122, 2000.
- [24] Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, Saurabh Bagchi, and Keith Whisnant. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579, June 1999.
- [25] D. Kunda and L. Brooks. Identifying and classifying processes (traditional and soft factors) that support cots component selection: a case study. *European Journal of Information Systems*, 9(4):226–234, December 2000.
- [26] Akos Ledeczki, Gabor Karsai, and Ted Bapty. Synthesis of self-adaptive software. In *Proceeding of the IEEE Aerospace Conference*, March 2000.
- [27] M. M. Lehman. Feedback in the software process. In *SEA Workshop: Research Directions in Software Engineering*, London, 1997. Imperial College.

- [28] Jiming Liu and Yi Zhao. On adaptive agentlets for distributed divide-and-conquer: A dynamical system approach. *IEEE Transactions on Systems, Man, and Cybernetics*, 32(2):214–227, March 2002.
- [29] Lennart Ljung. *System identification: Theory for the user*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.
- [30] Ying Lu, Tarek Abdelzaher, Chenyang Lu, and Gang Tao. An adaptive control framework for QoS guarantees and its application to differential caching services. In *10th IEEE International Workshop on Quality of Service*, pages 23–32, 2002.
- [31] N. Maiden and C. Ncube. Acquiring cots software selection requirements. *IEEE Software*, pages 46–56, March/April 1998.
- [32] Jim Melton. Xml-related specifications (sql/xml), August 2002. ISO-ANSI Working Draft.
- [33] M. Morisio and M. Torchiano. Definition and classification of cots: A proposal. *Lecture Notes in Computer Science 2255*, pages 165–175, 2002.
- [34] C. Ncube and N. Maiden. Guiding parallel requirements acquisition and cots software selection. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 133–140, 1999.
- [35] Peter Norvig and David Cohn. Adaptive software. *PC AI Magazine*, January 1997.
- [36] M. Ochs, D. Pfahl, G. Chrobek-Diening, and B. Nothhelfer-Kolb. A method for efficient measurement-based cots assessment and selection - method description and evaluation results. In *Proceedings 7th International Software Metrics Symposium*, pages 285–296, 2001.
- [37] S.B. Sassi, L.L. Jilani, and H.H.B. Ghezala. Cots characterization model in a cots-based development environment. In *Proceedings of the Software Engineering Conference*, pages 352–361, 2003.
- [38] R. Seacord, D. Mundie, and S. Boonsiri. K-bacee: Knowledge-based automated component ensemble evaluation. In *Proceedings of the 2001 Workshop on Component-Based Software Engineering*, 2001.
- [39] David C. Steere, Ashwin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Operating Systems Design and Implementation*, pages 145–158, 1999.
- [40] J. Voas. Disposable cots-intensive software systems. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 239–241, 2000.
- [41] David L. Wells and Paul Pazandak. Taming cyber incognito tools for surveying dynamic/reconfigurable software landscapes. In *Working Conference on Complex and Dynamic Systems Architecture (CDSA)*, Brisbane, Australia, December 12-14 2001.
- [42] D. Yakimovich, G. Travassos, V., and Basili. A classification of software components incompatibilities for cots integration. In *Proceedings of the 24th Software Engineering Workshop*, 1999.

- [43] I. Yen, J. Goluguri, F. Bastani, L. Khan, and J. Linn. A component-based approach for embedded software development. In *Proceedings of the 5th International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 402–410, 2002.
- [44] I. Yen, L. Khan, B. Prabhakaran, F. Bastani, and J. Linn. An on-line repository for embedded software. In *Proceedings of the 13th International Conferences on Tools with Artificial Intelligence*, pages 314–321, 2001.
- [45] C. Yonghao and B. Cheng. Facilitating an automated approach to architecture-based software reuse. In *Proceedings of the 12th IEEE International Conference on Automated Software Engineering*, pages 238–245, 1997.
- [46] G. Grau, J.P. Carvallo, X. Franch and C. Quer. DesCOTS: a software system for selecting COTS components. In *Proceedings of the Euromicro Conference*, pages 118-126, 2004.

Authors addresses:

João W. Cangussu

Department of Computer Science

University of Texas at Dallas

e-mail: cangussu@utdallas.edu

Kendra Cooper

Department of Computer Science

University of Texas at Dallas

e-mail: kcooper@utdallas.edu

Eric Wong

Department of Computer Science

University of Texas at Dallas

e-mail: ewong@utdallas.edu