# A Run-Time Adaptable Persistency Service using the SMART Framework

João W. Cangussu  Kendra Cooper  Eric Wong  Xiao Ma
Department of Computer Science
University of Texas at Dallas
Richardson-TX 75083-0688, USA
{cangussu,kcooper,ewong,xxm012000}@utdallas.edu

## Abstract

*Run-time adaptation is becoming an important feature to increase robustness, performance, and improve resource utilization. A system using a persistency data service can benefit from this feature by avoiding constraint violations. This aids in averting system failures and consequent data loss. The application and assessment of a run-time adaptable approach, the SMART (State Model Adaptive Run-Time) Framework, to a data persistency service is the subject of this paper. SMART is a component-based approach with adaptable features founded on the mathematics of control theory and fuzzy logic. The results of the study indicate that the use of SMART to the persistency data service avoided constraint violations in 95% of the cases over an execution period of more than eight hours. These results are encouraging and provide strong evidence of the advantages of using the SMART Framework.*

## 1 Introduction

Algorithms used for achieving data persistency may have a major impact on a system's response time performance, robustness, and memory requirements. A scenario where a system or application has a high failure intensity may lead to significant data loss if the persistency data service algorithm keeps data in memory for too long before saving it to the storage. Alternatively, an algorithm that immediately saves all modified data to the storage may lead to slow response time.

A customized persistency data service can achieve good results when designed for a specific and stable environment. However, adaptable systems have to be designed to cope with constantly changing environments or to run on distinct platforms. For example, a persistency service may be executing on a workstation with a powerful processor and ample memory or in a mobile unit with remote storage and subject to network availability and utilization and noise interference. Furthermore, extra constraints can be imposed on the system. For example, consider a security mechanism that is frequently scheduled to run in a real-time operating system with no virtual memory management. The mechanism demands 10% of the memory and must be able to run at any time. This requires that 10% of memory must always be available; the constraint applies to all applications executing in the system. This scenario not only demands run-time adaptation but can also benefit from prediction capabilities to avoid potential failures.

The SMART (State Model Adaptive Run-Time) Framework appears to be an answer for the issues stated above and its application to a persistency data service is analyzed here. SMART is a component-based approach founded on the mathematics of control theory and fuzzy logic. System identification techniques are used at run-time to tune a controller for the application based on its inputs, outputs, and resource usage. This allows for control and prediction of the future behavior of the variables under consideration. Once a prediction is made that a constraint violation will occur, a new component that better copes with the current system status is selected.

In this work, a component is "a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files" [1]. The overall architecture of the SMART framework is a tailored process control style [2]. A future line of investigation is to extend SMART to a component based middleware solution (e.g., Distributed Component Object Model [3], Enterprise Java Beans [4], CORBA Component Model [5]).

Specifications of the available components are stored in a repository; fuzzy logic and multi-criteria decision making algorithms are used to select the component that better copes with the current system resources. In spite of the overhead created by the execution of support tasks (resource monitoring, controller tune up, etc.), the results of this study

demonstrate the use of SMART to control the execution of a persistency data service avoided constraint violations in 95% of the cases over an execution period of more than eight hours. These encouraging results provide strong evidence of the advantages of using the SMART Framework.

The remainder of this paper is organized as follows. Control theory and fuzzy logic, the theoretical foundations upon which the SMART Framework is constructed, are briefly described in Section 2. An overview of the SMART Framework is the subject of Section 3. An adaptable persistency data service using the SMART Framework is described in Section 4 along with the collected results. Section 5 presents some related work and concluding remarks are presented in Section 6.

## 2 Theoretical Foundations

The mathematical foundation of the SMART Framework is presented next. The control theory description focuses on state models and system identification techniques in Section 2.1, while fuzzy number and fuzzy multi-criteria decision making algorithms are overviewed in Section 2.2.

### 2.1 State Models and System Identification

Linear state feedback models have provided useful representations for large classes of engineering, biological, and social processes [6, 7]. 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 $\begin{cases} \dot{x}(t) = Ax(t) + Bu(t) \\ y(t) = Cx(t) + Du(t) \end{cases}$, 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." [8]

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 presented and a model can be inferred from the observations.

A number of techniques, such as Least-Square and Markov Parameters [8] are available to identify state space models but their discussion is beyond the scope of this paper. In order to implement a fast prototype of SMART, the Least-Square identification procedure available in MATLAB was chosen.

### 2.2 Reasoning about Uncertainty: Fuzzy Logic

The uncertainty in the non-functional behavior of components needs to be captured and reasoned about in the SMART Framework. There are numerous ways to represent uncertainty including possibility measures, probability measures, belief functions, and others [9]. Here, an overview of the underlying theory of possibility measures, fuzzy sets and logic, is presented. This is the mathematical foundation used to specify and reason about the selection of components in SMART. Two main characteristics of fuzzy systems, described by Carlson and Fuller [10], provide the motivation for its use. The first is that "fuzzy systems are suitable for uncertain or approximate reasoning, especially for systems with mathematical models that are difficult to derive". The second is "that fuzzy logic allows decision making with estimated values under incomplete or uncertain information". These two characteristics are inherent to specification and reasoning about the non-functional behavior of components. Fuzzy multi-criteria decision making algorithms, originally developed for use in business management decision making, are available [10]. One such algorithm is initially adopted in this research; potential improvements in the algorithm are subjects of future investigation.

There are different kinds of fuzzy numbers, such as triangular and trapezoidal. For example, a triangular fuzzy number with peak, or center $a$, left width $\alpha < 0$, and right width $\beta > 0$ has the form:

$$A(t) = \begin{cases} 1 - \frac{(a-t)}{\alpha} & if\ a - \alpha \leq t \leq a \\ 1 - \frac{(t-a)}{\beta} & if\ a \leq t \leq a + \beta \\ 0 & otherwise \end{cases} \quad (1)$$

and we use the notation $A = (a, \alpha, \beta)$. It can be easily verified that $[A]^\gamma = [a-(1-\gamma)\alpha, a+(1-\gamma)\beta]$, $\forall \gamma \in [0, 1]$. The support of A is $(a - \alpha, b + \beta)$. A triangular fuzzy number with center $a$ may be seen as a fuzzy quantify with the meaning "x is approximately equal to $a$"

Fuzzy sets can be used to represent linguistic variables. A linguistic variable is characterized by a quintuple (x, T(x), U, G, M) in which x is the variable name; T(x) is the set of terms for x (i.e., the set of linguistic values such as "slow" or "fast"); U is the universe of discourse, G is a syntactic

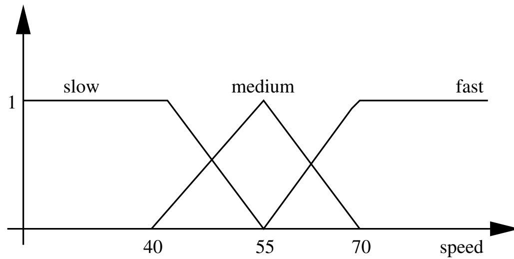


**Figure 1.** Fuzzy partition.

rule for generating the names of values of x; and M is a semantic rule for associating each value to its meaning.

For example, the linguistic variable "speed" (refer to Figure 1) has the terms "slow", "medium", and "fast". These can be characterized as fuzzy sets whose membership functions are:

$$slow(v) = \begin{cases} 1 & if\ v \leq 40 \\ 1 - \frac{v-40}{15} & if\ 40 \leq v \leq 55 \\ 0 & otherwise \end{cases} \quad (2)$$

$$medium(v) = \begin{cases} 1 - \frac{|v-55|}{30} & if\ 40 \leq v \leq 70 \\ 0 & otherwise \end{cases} \quad (3)$$

$$fast(v) = \begin{cases} 1 & if\ v \leq 70 \\ 1 - \frac{70-v}{15} & if\ 55 \leq v \leq 70 \\ 0 & otherwise \end{cases} \quad (4)$$

Linguistic variables can be defined for each non-functional property of interest for the components (e.g., performance, memory requirements, etc.).

# 3 The SMART Framework

The SMART Framework is based on the use of control theory to manage the adaptation issues when applications/environments have resource constraints and multiple design choices are available to cope with the constraints and the constantly changing environment. The use of system resources is unlike to present a linear behavior. However, its dominant behavior is expected to be captured by a linear approximation. This approximation has been demonstrated to be general enough for such representations [11]. After a model is specified, feedback control can be applied to predict and regulate (i.e., select alternative design choices) the behavior of the system.

The overall goal of the SMART Framework is to provide an environment for the design of 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. 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 use of feedback control theory to support the continuous monitoring and control of the system in a dynamically changing environment. The second area is the effective specification, matching, and selection of 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. Once identified the components can be swapped into the system to better meet the needs of the users. The integration of these two areas of research are reflected in the architecture of the SMART Framework. In general terms, the implementation can be done using any programming language if the application does not require new components to be added at run time. This is the case for the persistency service application where all components are available at compilation time. If dynamic loading is required, then the SMART Framework needs to make use of a dynamic programming language such as Java or Dylan [12].

## 3.1 The SMART Framework Architecture

As shown in Figure 2, the architecture of the SMART Framework is a tailored process control architectural style composed of the component specification repository, actuator, pre-replacement, current component, system identification module, resource monitor, and controller. The system identification module receives data from the resource monitor and generates/updates the controller. Based on the current status of the system (also provided by the system resource) the controller predicts the future behavior of the application. If a constraint violation is predicted, then the actuator accesses the component specification repository to select the component that is better suited to handle the violation. In case some glue code is needed, the pre-replacement module prepares the component before the actual swap occurs. The architectural components and their relationships are described below.

**Component Specification Repository**: This repository contains the specification of the alternative components that are available to address a problem in distinct scenarios. Each component is specified in fuzzy logic to describe its functional properties, non-functional properties (performance, memory usage, interface requirements, etc.), the composition of the solution (a solution may be composed

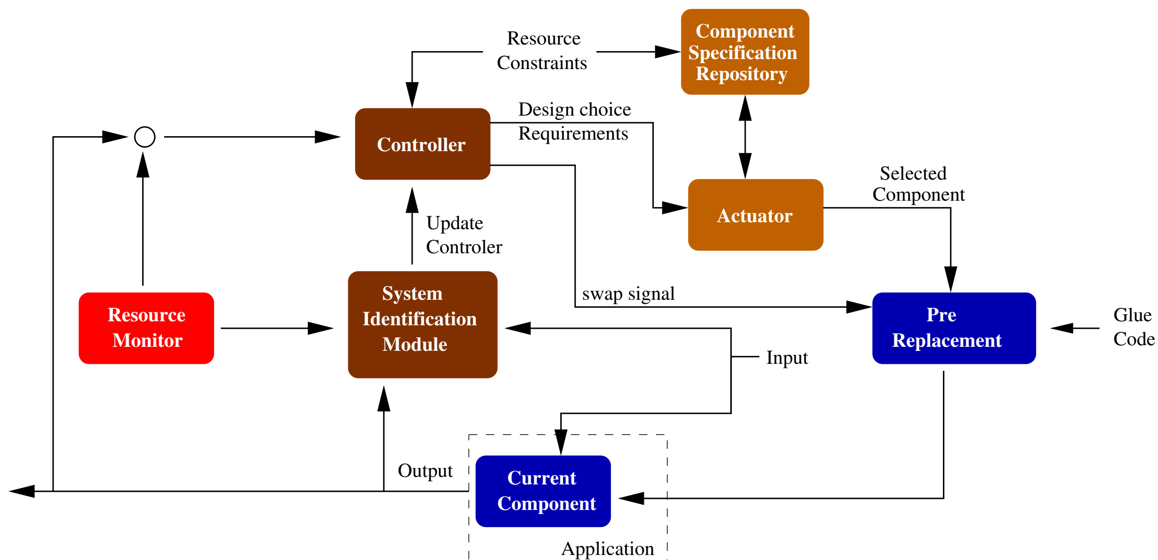


**Figure 2.** Overview of the SMART Framework architecture.

of one or more components), and the location of the code in the system.

The behavior of the components is mapped to an n-dimensional solution space, where n is the number of quality attributes considered. As a simple example, consider two quality attributes: response time and memory. Response time has linguistic variable values very slow, slow, medium, fast, and very fast; memory has values low, medium and high. The solution space is structured into overlapping quality zones using the combinations of the variable values (e.g., memory is medium, response time is fast is one quality zone). The solution spaces are for specific execution environments.

The component specification repository is being implemented using MATLAB and its fuzzy logic toolbox. Currently, a simplified version in C is available.

Each alternative solution (i.e., a collection of one or more components) must be smoothly incorporated to the system. If recompilation and relinking are not an option, then techniques such as structural conformance, delegation and wrapping are some alternatives to be addressed to achieve this goal [13].

**System Identification Module**: In order to use control theory [6] one needs a mathematical model (system of equations) that relates the inputs and outputs of the system. Since the components and their interactions affect the systems resources, as a result, a specific model is unlikely to accurately capture the dominant behavior of all the components. However, system identification techniques [8] allow for the dynamic determination of tailored models for a specific component/system. As stated in Section 2.1, a least-square identification procedure from MATLAB is used in this module.

This part of the SMART Framework is similar to the work done for autonomous agents [14]. 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 subsequently updated. System identification techniques such as Markov Parameters and the Least Square approach can be used to implement this module.

**Controller**: The controller is generated/updated by the **System Identification Module**. The controller predicts when a bottleneck constraint is expected to occur and determines the latest possible binding time to correct the problem. The current state of the system is used as an initial condition and the predictions are made based on the behavior of the model for a desired time window or steps ahead. Clearly, the number of steps ahead to be predicted affects the accuracy of the prediction. This prediction allows for the minimization of the overhead costs associated with component swap. The controller uses the *resource constraints* to check if the prediction violates any specified constraint.

**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 choice that is running. It receives the input and passes in-

formation (output) to the **Controller** and the **System Identification Module**.

**Actuator**: The actuator selects one or more components specified in the component specification repository. The selection algorithm is a fuzzy multi-criteria decision making algorithm, as described by Carlsson and Fuller [10].

The actuator uses the desired system behavior from the controller that should be met to achieve the system's integrity, performance, and consistency. To the best of our knowledge, fuzzy multi-criteria decision making algorithms have not yet been applied in component-based research.

**Pre-Replacement**: This module prepares the component selected by the actuator to replace the current one. It executes any finishing task, for example flushing the memory before the swap from $P_2$ to $P_1$ can be done in the persistency service application described in Section 4. It also inserts any necessary glue code and links it to the component. The **Pre-Replacement** then waits for a signal from the controller determining the exact time when the swap should occur. This technique improves the response time performance by preparing the component in advance.

A prototype for the SMART Framework has been implemented for a customized application. The goal, as described in Section 4, is to show the applicability of the adaptive framework. However, we are aware of implementation issues that need to be addressed. The major concern is to decrease the overhead created by the use of the framework. The resource monitor, the system identification module, the controller, and the actuator create overheads at execution and interface level. Alternatives to decrease the overhead are briefly described in Section 4.

# 4 An Adaptable Persistency Data Application

The results of applying the SMART Framework to an example system are presented here. The example is an application 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 is 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 as presented by Govi [15]. 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.

## 4.1 Persistency Service Options

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.

The first option provides a solution that uses minimal memory in the TDS. Reading and writing to the disk creates slower response times, however, it also improves data integrity as at most changes to one item may be damaged or lost if the application terminates unexpectedly (i.e., crashes). 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 from the PDS, $e_{iPDS}$. 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.

The second option provides a solution that uses more memory in the TDS. However, fewer reads and writes to the disk results in a faster response time. The data integrity is reduced compared to the first option, as changes to multiple entities may be damaged or lost if the system fails. In summary, this option provides a solution that can be characterized as high memory, fast response time performance, and moderate data integrity. Thus, these solutions can be placed into *quality zones*, or solution spaces, as:

- low memory, slow response time performance, high data integrity for $P_1$
- high memory, fast response time performance, moderate data integrity for $P_2$

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. Using this data, the solutions are manually mapped into a *quality zone*. Due to the uncertainty in the boundaries of the overlapping solution spaces and the solutions, fuzzy logic is used to represent and reason about them. The trade-off between memory usage and performance (execution time) is considered here.

Component $P_1$ has a static memory allocation amounting to approximately 0.2% of the total memory of the system. Component $P_2$ allocates memory dynamically with a lower bound of 0.2% and, theoretically, a 100% upper bound of memory usage. Regarding to execution time, as described before, $P_2$ has a better performance than $P_1$. However, the execution time for $P_2$ depends on the hit ratio, i.e., the percentage of times the data needed is already in the TDS.

## 4.2 Component Evaluation

The behavior of the components are measured at runtime. Each component is manually instrumented to "dump" the desired data to a file. No specific application is used in this case. The data are used to derive a specification of the components stored in the component specification repository. For example, both components are executed to obtain the total time and total memory execute measurements. Each component is executed 100 times; each execution has 200 client request modifications.

The idea is to measure the time each component takes to execute the same number of requests. Furthermore, since the hit ratio[1] influences the performance of $P_2$, this component is evaluated using a) fixed hit ratios ranging from 0.3 to 0.9, and b) a uniform randomly distributed hit ratio. It was observed, in all the cases, that $P_2$ has a faster average execution time than $P_1$. More specifically, $P_2$ has an average performance improvement over $P_1$ ranging from 25.9% (for a 0.9 hit ratio) to 12.3% (for a 0.3 hit ratio). The experiments with a random hit ratio produced 18% improvement in performance. Therefore, these components display a clear memory/performance trade-off.

[1] The hit ratio here refers to the Temporary Data Store and not the cache memory used by the Operating System.

## 4.3 Adaptive Application Design

An application which adapts to the changing amount of memory in the environment is presented here. 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 **predicted** by the SMART Framework.
- Swap from $P_1$ to $P_2$: occurs when more than 5% of memory is available in the system.

The use of the SMART Framework allows for the execution of the persistency service application with the swap between components $P_1$ and $P_2$. The first prototype of the framework has been implemented in C language (Application, Resource Monitor, Pre-replacement, Actuator modules), and MATLAB scripts (System Identification and Controller modules). The prototype is going to be refined as the work progresses. For example, there is no need for "glue code" in the current application and the "Pre-Replacement" module is implemented as a simple procedure. The "Actuator" block is going to be changed to implement the fuzzy multi-criteria decision making algorithm.

The interface between the C program ("Current Component", "Resource Monitor", and "Actuator") and the MATLAB scripts ("System Identification Module" and "Controller") is implemented using a file buffer. The input (number of client requests), output (number of saved requests), and resources (memory available and memory used by the process) are written to a file buffer. The scripts in MATLAB read this information from the buffer and use them to make a prediction of the amount of memory needed in the next time period (5 seconds in this case). If $P_2$ is executing and a prediction that memory usage is going to cross the threshold is made, then MATLAB writes to another file buffer and the "Actuator" thread in the C program reads this information and make the swap from $P_2$ to $P_1$. Once the available

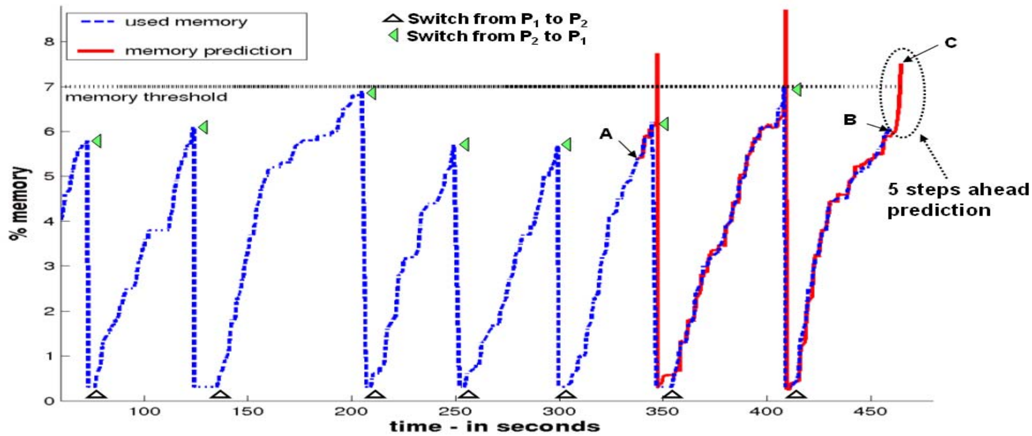


**Figure 3.** Snapshot of the memory used by the persistency service application and the corresponding predictions made by SMART. The framework signals swaps from $P_2$ to $P_1$ (for example at time t=125s) and from $P_1$ to $P_2$ (for example at time t=133s). As can be noticed the use of SMART to signal the swap from $P_2$ to $P_1$ has prevented the constraint violation over the whole period. A new violation within the next 5 seconds is predicted, from point B, and another swap from $P_2$ to $P_1$ is signaled to avoid the constraint violation. The actual swap occurs the next time the thread in charge of swapping is scheduled. To decrease the overhead, only the last 200 data values between points A and B are used in the prediction of the next 5 steps (between points B and C).

memory goes above the 5% limit, a swap from $P_1$ to $P_2$ is performed.

### 4.4 Adaptive Application Evaluation

Figure 3 shows a snapshot of the execution of the persistency data service. It can be observed that the memory constraint is not violated and the swaps are done in time. The figure shows that another swap, within the next five seconds from point B, has been predicted to avoid crossing the memory threshold. It can also be noticed from Figure 3 that in almost all the cases, as soon as $P_1$ starts to execute a new swap to $P_2$ (represented by "empty triangles") is detected. The reason for this is because more than 5% of memory is available.

Another important aspect to be noticed from Figure 3 is that only part of the data is used to make the prediction. The more data is available the more accurate are the predictions. However, the overhead also increases with more data and balance between accuracy and overhead is kept by using only the last 200 data points, found between points A and B in the figure, to make the five steps ahead prediction for point C. The data points are from the current execution and not from a historical data set.

The persistency service application and the associated MATLAB scripts were 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 $P_1$. This would hypothetically crash the persistency service application, but for experimental purposes, only the number of failed swaps are recorded. Though the threshold was exceeded 37 times, it should be noticed that the amount of memory used is on average only 0.36% above the 7% threshold as can be noticed in Figure 4.

The detected failures are due not to wrong predictions but in almost all the cases to late reaction time. That is, SMART has predicted the constraint violation before it happened but due to the overhead in the prediction, the overhead in the communication between MATLAB and C, and the scheduling of the "Actuator" thread, the swap is not done in time and the used memory exceeded the threshold. The scheduling problem and the communication overhead can be minimized by the use of interprocess communication. The overhead in the prediction could be also minimized by having a compiled version of the MATLAB scripts. However, not all system identification functions in MATLAB can be converted to C code. A partial solution to decrease the overhead is the decomposition and conversion of parts of the scripts. Another alternative to decrease the number of failures is to increase the number of steps ahead used in the prediction. This would give more reaction time to the persistency service 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.

In general, we conclude that this example of the use of

the SMART has produced encouraging results showing the applicability and feasibility of the framework. We are aware that more validation is needed and a thorough validation plan has already been delineated [16].

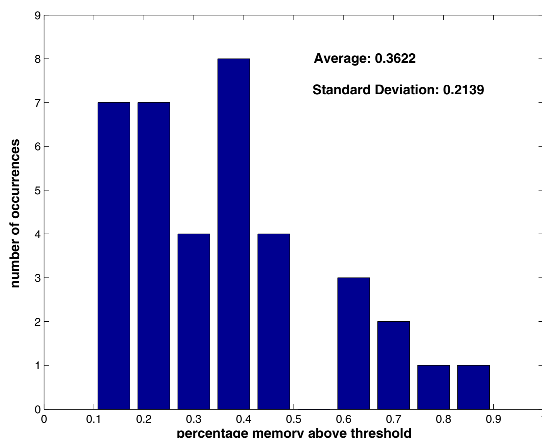


**Figure 4.** Histogram of the percentage of memory violations (Persistency Service failure cases).

## 5 Related Work

The SMART Framework is an adaptive approach based on the selection of components specified in our repository. Therefore, the topics component-based architecture and adaptive systems are the most related to SMART. Here we present related work on adaptive systems in Section 5.1. Work related to component-based software design using repositories (as by Yen , Bastani, et. al [17]; Seacord, Mundie, and Boonsiri [18]; and Yonghao and Cheng [19]) is available elsewhere [20].

### 5.1 Adaptive Systems

Self-adaptive systems has been proposed for specific domains such as QoS [21], ad-hoc network management, traffic control, and signal processing [20]. 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 [20]. Approaches that are not domain specific have also been proposed [22]. Many of successful approaches are based on the sound theory of feedback control [21, 20]. Decision theory is another technique used in adaptive systems providing mathematical tools, such as game theory and subjective probability, to address decision making problems in the presence of uncertainty. A large body of methods have also resulted from the Dynamic Assembly for Systems Adaptability, Dependability, and Assurance (DASADA) project sponsored by DARPA.

Two approaches are more closely related to the SMART Framework: Containment Units [22] and the QoS Framework [21]. 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 former is also defined according to nonfunctional requirements, such as time, memory, and sensors [22]. The architecture of a *Containment Unit* is composed of the "Operational Component", the "evaluator", the "change agent", the "Adapter", and the "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 [22]. Similarly to SMART, *Containment Units* is a component-based framework for dynamic adaptable systems. Both approaches present a component swap as an alternative to better use system resources. The major difference between the two approaches is that SMART can predict and consequently prevent constraint violations. In addition, the system identification module in SMART allows a dynamic customization of the models making SMART a more flexible approach.

The work done on adaptive systems for QoS guarantees [21] 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. The SMART Framework also has a similar architecture to the one presented in the QoS Framework. 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 pointed out in Section 6, SMART is a more generic approach by allowing the specification of the system constraints and their relation to the alternative choices.

### 5.2 Adaptive Persistence

A large body of work is available in the literature on data persistency covering issues such as load balance, replication, and migration [23]. With respect to the SMART

Framework, research in the run-time adaptability of data persistency is of great interest. Work done by Ortin and Cueva in the nitrO Reflective Platform [24] is briefly described next.

NitrO is a non-restrictive reflective platform that has been implemented using Python. The non-restrictive feature of nitrO comes from its language and application independence characteristics that do not rely on any specific interpreter protocol. The selection of Python was due to the reflective features such as introspection, structural reflection, and dynamic evaluation [24]. The adaptive characteristics of nitrO are based on changing the semantics of the programming language at run time (not only the symbol table, but the language specification itself can be changed at run-time). In this way, an application can adapt the structure or the behavior of another application. A persistence subsystem has been implemented using nitrO. Adaptability is achieved by dynamically updating the policies and/or storage methods used in the system. Three reference storages and two update policies exist. The switch from one policy/storage to another is done based on the application structure in a programmatic way. This represents a major difference between the adaptation mechanism of SMART and nitrO since the former uses system resource and the input/output of the application to dynamically adapt the system at run-time.

## 6 Concluding Remarks

An empirical assessment of a run-time adaptable persistency service is presented in this work. The results of the study demonstrate the use of the SMART Framework avoided constraint violations in 95% of the cases over an execution period of more than eight hours.

In addition to the advantages of many adaptive system approaches (self-adaptation, robustness, better resource usage, etc.), the novel application of control theory and fuzzy logic in the SMART Framework provides the following two distinguishing advantages:

- 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 identifies these relationships and therefore allows its application to a variety of different domains. In addition, FMCDM algorithms may be applied to a wide variety of components.

- 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 spite of the advantages above, 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 X 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. Better implementation and design choices will definitely reduce the overhead. Consequently this reduces the overall swap time and the number of constraint violations.

SMART is a new technique to handle the dynamic behavior that is rapidly becoming part of many companies daily concerns and the results of its application to a persistency data service are evidence of its applicability and accuracy. SMART has also been applied to another application performing multiplications of large sparse matrices [25] with similar convincing results. Furthermore, a complete validation of the framework is planned with self and cross-evaluation for additional systems such as data compression and image interpolation.

## References

[1] "Omg unified modeling language specification." The Object Management Group, Inc., March 2003. Version 1.5, formal/03-03-01.

[2] M. Shaw and D. Garlan, *Software Architecture: Perspectives on Emerging Discipline*. Prentice Hall, Upper Saddle River, New Jersey, 1996.

[3] "Microsoft com technologies - dcom." Microsoft Corporation, 2000. available at http://www.microsoft.com/com/tech/dcom.asp.

[4] "Enterprise javabeans technology." Sun Microsystems, 2001. available at http://java.sun.com/products/ejb.

[5] "Corba components model." Object Management Group. available at http://www.omg.org/cgi-bin/doc?ptc/99-10-05.

[6] D. G. Luenberger, *Introduction to Dynamic Systems: Theory, models and applications*. New York: John Wiley & Sons, 1979.

[7] J. W. Cangussu, R. A. DeCarlo, and A. P. Mathur, "A formal model for the software test process," *IEEE Transaction on Software Engineering*, vol. 28, pp. 782–796, August 2002.

[8] L. Ljung, *System identification: Theory for the user*. Englewood Cliffs, New Jersey: Prentice-Hall, 1987.

[9] J. Y. Halpern, *Reasoning about Uncertainty*. MIT Press, 2003.

[10] C. Carlsson and R. Fuller, *Fuzzy Reasoning in Decision Making and Optimization*. Physica-Verlag Heidelberg, 2002.

[11] Y. Lu, A. Sexana, and T. F. Abdelzaher, "Differentiated caching services: a control-theoretical approach," in *Proceedings of the 2001 International Conference on Distributed Systems*, pp. 651–622, 2001.

[12] A. L. M. Shalit, *Dylan Reference Manual*. Addison-Wesley, 1996.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] J. Liu and Y. Zhao, "On adaptive agentlets for distributed divide-and-conquer: A dynamical system approach," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 32, pp. 214–227, March 2002.

[15] G. Govi, "Data service documentation." http://pool.cern.ch/storage/doc2/DataSvc.pdf, 2003. Version 0.5.

[16] J. W. Cangussu, K. Cooper, E. Wong, and X. Ma, "An adaptive persistency service using the SMART framework," Tech. Rep. UTDCS-05-04, University of Texas at Dallas, Department of Computer Science, February 2004.

[17] 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*, pp. 402–410, 2002.

[18] 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.

[19] 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*, pp. 238–245, 1997.

[20] J. W. Cangussu and K. Cooper, "A new approach for the design and control of adaptive systems," Tech. Rep. UTDCS-21-03, University of Texas at Dallas, Richardson-TX, USA, May 2003.

[21] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao, "An adaptive control framework for qos guarantees and its application to differential caching services," in *10th IEEE International Workshop on Quality of Service*, pp. 23–32, 2002.

[22] J. M. Cobleigh, L. J. Osterweil, A. Wise, and B. S. 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)*, (Charleston, SC), pp. 159–165, November 2002.

[23] M. Kasbekar, S. Yajnik, Y. H. Reinhard Klemm, and C. Das, "Issues in the design of a reflective checkpointing library for c++," in *18th IEEE Symposium on Reliable Distributed Systems*, (Lausanne, Switzerland), October 1999.

[24] F. Ortin and J. M. Cueva, "The nitro reflective platform," in *International Conference on Software Engineering Research and Practice (SERP 2002)*, (Las Vegas, USA), 2002.

[25] J. W. Cangussu, K. Cooper, and C. Li, "A control theory based framework for dynamic adaptable systems," in *19th Annual ACM Symposium on Applied Computing, SAC 2004*, (Nicosia, Cyprus), ACM, March 14 -17 2004.