

A Model Integrated Framework for Designing Self-managing Computing Systems*

Jia Bai
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee 37235
jia.bai@vanderbilt.edu

Sherif Abdelwahed
Department of ECE
Mississippi State University
Mississippi State, Mississippi 39762
sherif@ece.msstate.edu

ABSTRACT

This paper presents a model integrated framework, referred to as the Automatic Control Modeling Environment (ACME), to facilitate the use of control-based technology for self-management in computation systems. ACME is a domain-specific graphical modeling environment with automated synthesis tools. The framework allows domain engineers to develop models for general computation systems and to capture their performance requirements and operational constraints. The framework can automatically generate executable codes for the controllers based on the given system model and specifications. A case study of an online power management is used to demonstrate the application of ACME.

Keywords

self management, predictive control, model-integrated computing

1. INTRODUCTION

Control-theoretic strategies have been recently applied successfully for the design and verification of various adaptive resource management schemes in computation systems. If system dynamics is precisely modeled and changing environmental parameters are accurately estimated, appropriate run-time control algorithms can be effectively developed to realize system self-regulation and achieve desired Quality of Service (QoS) objectives. Examples of control-based resource management strategies include task scheduling [11, 14], QoS guarantees in web servers [10], resource allocation control [5, 13], network flow control [8], and power management [3].

Domain engineers and developers of computing systems, however, may not have the background to apply and implement control-based methods. To facilitate the use of control-based

*This work was supported in part through a grant from the NSF SOD program, contact number: CNS-0804230

techniques, a middleware QoS-control architecture is proposed in [15] to provide software performance assurances based on linear feedback control theory. Also in [12], a model-based design framework, referred to as the Dynamic QoS Modeling Environment (DQME), is developed to achieve end-to-end QoS management in computing systems using model-based predictive control strategies.

In this paper, we present a model-based design framework that facilitates the design of general control-based adaptation components for a general class of computational systems. The framework includes a generic control library from which a controller can be selected and parameterized for a given system and operation settings. The framework allows the user to develop formal models capturing relevant aspects of the system behavior as well as its performance specification. The models are then used by an interpreter to automatically generate executable codes for an appropriate control module. This framework is referred to as the Automatic Control Modeling Environment (ACME). The framework is based on the Generic Modeling Environment (GME) [9], a meta-programmable toolkit, which allows for easy creation of domain specific modeling languages and environments.

The ACME framework allows the design and specification of general control-based QoS adaptation policies. As a specific implementation case study, we consider the limited lookahead control (LLC) approach proposed in [2, 6, 7] for resource management of distributed system applications. In this approach, control actions are computed to optimize system behavior for pre-specified QoS criteria over a limited look-ahead prediction horizon. Control objectives are represented explicitly in the form of a multi-variable optimization problem, and solved at every control step.

The rest of this paper is organized as follows: Section 2 provides an overview of key concepts of the ACME language. Section 3 shows the basic design components of the ACME meta-model. Section 4 discusses the ACME interpreter which translates models into executable codes. A case study using the LLC approach is presented in Section 5. Conclusion and future research are discussed in Section 6.

2. ACME OVERVIEW

Effective self management requires the ability to monitor and tune system variables that affect various QoS related parameters. Those parameters are often inter-dependent, i.e. modifications made on one may affect others. Also, op-

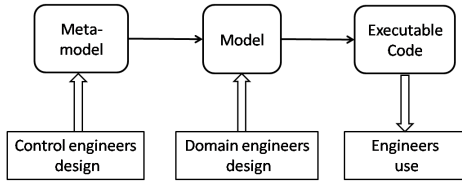


Figure 1: ACME design process

erational constraints such as resource limitations and safety margins impose additional requirements on the system. The inter-dependencies and constraints need to be effectively captured for a self-management design. In addition, future variations in the system components and structure need to be considered as well to guarantee the system performance. Control-based techniques have proven to be effective in addressing the above requirements for self-management design and in addition they can provide performance guarantees under given operating conditions. However, the adoption of such techniques remains limited due to lack of tools and libraries that facilitate the control-based design for design engineers.

To address this problem, we propose in this paper the ACME framework. The ACME is control-theory oriented framework aimed at providing effective self management for computation systems. Fig. 1 shows the develop process of the ACME. Structurally, the ACME is composed of three main aspects. One is the architecture structure, in which high level components and their interconnections are defined. Another is the data collection entity, which is responsible for collecting system measurements corresponding to the model variables. The third one is the system dynamics structure used for capturing the system model, specifications and operation constraints, as well as providing modules for estimating system future variations and tuning system variables with respect to operational variations and constraints.

Although LabView and MatLab have control toolkits with similar interface to ACME, the ACME can generate various executable codes, like C++, XML, Python, or even MatLab codes upon requests based on the graphical models. More importantly, control engineers can easily modify the modeling structure and specifications when necessary by updating meta-models.

The following subsections describe the semantic intent of the key modeling components in ACME. In this paper, to enhance readability, the following font-based notations are adopted: “components” used for the main components of the meta-model, “connections” used for the connections between the components, “visible” used for the visibility aspects of models, and “attribute” used for the component attributes.

2.1 Architecture

The architecture in the ACME captures the main structure of the whole system. It contains any of the components in the self-management design, as well as the connections between the underlying ports of these components. From the architecture level of view, the designer can construct the high-level components of the system and define the connec-

tions between them. The details of these components are encapsulated in the underlying substructures, which have their own internal descriptions.

2.2 Data Collection

The data collection entity contains all the system variables. In practical systems, some of the system variables can be measured directly while others cannot. In some situations, system variables that cannot be measured can still be calculated based on the measured variables using observers. In other applications, future values of certain system variables need to be estimated. ACME distributes the data collection tasks to three different entity models as follows. First, a **Sensor** model reads in all the measurable data, which include environment inputs, observable system states, and system outputs. Latency, bandwidth, and CPU utilization are examples of observable system states for some class of systems. Second, to calculate the system states that cannot be observed directly, an **Observer** model collects all the related variables and computes the system states by association equations. Third, an **Estimator** model uses the latest and historical sample data to estimate future system variables. An example of implemented estimators in ACME is the autoregressive moving average (ARMA) estimator. In general, the user can choose estimators that best fit the system configuration from an estimator library in ACME.

2.3 System Dynamics and Adaptation

In the ACME framework, the system dynamics is a schematic description that captures the known or inferred behavioral properties of a computation system. The system dynamics is used for the design and verification of the self-managing structures.

The system adaptation specification represents the configuration of a controller module chosen from the control library available in the ACME. For example, the LLC controller can be selected as the system adaptation module, and can be configured by identifying the look ahead horizon, the possible control input set, and a utility function that characterizes each point in the QoS space with a utility value (or cost). The LLC utilizes these specifications to manage the system at run-time by optimizing the underlying system utility within the constraints posed by certain operational requirements.

3. ACME META-MODELS

This section introduces the ACME meta-models corresponding to the basic aspects of a self-management design specification. The aim of this modeling approach is to capture the system design in a modular component-based form that can be easily accessible to the system designer. For example, the **Estimator** model discussed in the previous section can be added to the architecture as a high-level component, parameterized, and connected to other model blocks in the architecture through their available ports. In the following subsection we presents the ACME meta-model, which is expressed with a stereotyped UML class-diagram notation. The stereotypes including `<<Model>>`, `<<Atom>>`, `<<Connection>>`, etc., express the binding of the abstract syntax to the concrete syntax implemented by the GME environment. Details of the concrete syntactic constructs

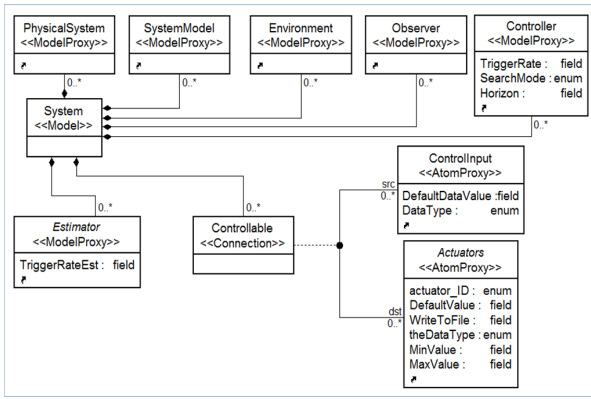


Figure 2: meta-model of the architecture modeling

supported by the GME environment are presented in [1, 9]. The sub-languages that constitute the ACME language are addressed below.

3.1 Architecture Models

The architecture stereotyped as a folder, contains a **System** model that collects all necessary parts of a system, each of which encapsulates its local components. In a distributed system, such as a web-server, a system involves multiple subsystems, each of which has independent local controllers with different performance requirements; also, a global controller addressing system-wide performance requirements will be constructed for the system, managing the interaction between the local controllers.

This model expresses the general structure of the overall system. Fig. 2 shows the meta-model of the architecture modeling sub-language. Note that the meta-model figures only show the main models, while other models are diminished in gray for simplification. The UML notation for containment is a line connecting an object to its container, with a small black diamond on the "container" end of the line. So **PhysicalSystem**, **SystemModel**, **Environment**, **Observer**, **Controller**, and **Estimator** are all key components which can be contained in the **System**.

The connections in the architecture define data transportation between models. As shown in Fig. 2, the **System** also contains a connection **Controllable**. The small black dot associates the connection with two endpoints **ControllInput** and **Actuators**, which act as ports of the high-level components, while the connection is directed from "src" to "dst". Similarly, signals in the **Environment** models can be sent to the **Estimator** models by **SensorToEst** connection, to the **Observer** models by **Measurement** connection, or to **SystemModel** by **SensorConn** connection; estimated variables can be sent from the **Estimator** models to the **SystemModel** through **EstSignalOut** connection; ports of system states in different blocks can be connected to each other by **SystemStateConn** connection, as can of control inputs with **ControlInputConn** connection.

3.2 Data Collection Models

All basic data types used in the meta-model like the **ControllInput** are first defined in a component paradigm. Sys-

temState, **SystemOutput**, and **ControllInput** are basic types of variables for control systems. **ControllInput** and **SystemState** represent the control inputs and system states respectively. **SystemState** and **ControllInput** can be used in the **Observer**, **SystemModel** and **Controller** models, while **SystemOutput** is used in the **Observer** only. Composite data types can be defined and modified only in the component paradigm, since data used in all the other places are proxies of the data in the component. The following models are used to get the values of the data proxies. The data types are often defined with their attributes, some examples of the attributes are name, type, IP address, and speed in a configured network system. In Fig. 2, **ControllInput** has two attributes *DefaultValue* and *DataType* in the lower half of the class rectangle.

3.2.1 Environment Model

The operation plants involved in certain environment always interact with the environment. The **Environment** model then represents the operation environment. In real time applications, the **Environment** only contains **Sensor** models to measure relevant environment variables from the real environment; in the simulation application, environment is simulated and environment variables are generated by the methods defined in a data generation library. For example, in the library, model **Reader** reads in data from local files, and **Generator** model can generate uniform distributed numbers. The generated data are then sent to other components via **Sensor** models.

3.2.2 Estimator Model

The **Estimator** model can be selected from an estimator library, where different estimators like ARMA filters and Kalman filters are included. For example, we use an ARMA filter to estimate the environment parameter such as future data arrival rate $\hat{\lambda}(k+1)$. Given the arrival rate $\lambda(k)$ at time k and the mean $\bar{\lambda}$ of past observations over a specified window size of m , the estimate rate for $k+1$ is:

$$\hat{\lambda}(k+1) = (1 - \sum_{i=0}^{m-1} \beta_i) \bar{\lambda} + \sum_{i=0}^{m-1} \beta_i \lambda(k-i), i \in [0, m]$$

where the gain β determines how the estimator tracks variations in the observed arrival rate. The ACME uses two kinds of models to represent the ARMA filter. The **HistAve** model specifies $\bar{\lambda}$, and its attribute *HistWindowSize* defines m . The **OrderedIndiv** model specifies the $\lambda(k-i)$, and its attribute *HistIndex* defines i (eg. a *HistIndex* of 1 represents the $(k-1)$ th observed data). Both models have *Parameter* attributes defining the gains $(1 - \sum_{i=0}^{m-1} \beta_i)$ and β_i respectively.

3.2.3 Observer Model

The **Observer** model calculates unobservable system states using measurable variables and parameters if the underlying functions are available. All the needed variables like **SystemOutput**, **Variable**, **ControllInput**, and **SystemState** are read in the **Observer** to the **Function** models to calculate the unknown values. Finally, **SystemStates** hold the computed data and assign them to other models.

3.3 Controller Model

The **Controller** model specifies the parameters of the controller design, and Fig. 3 shows the meta-model of the LLC

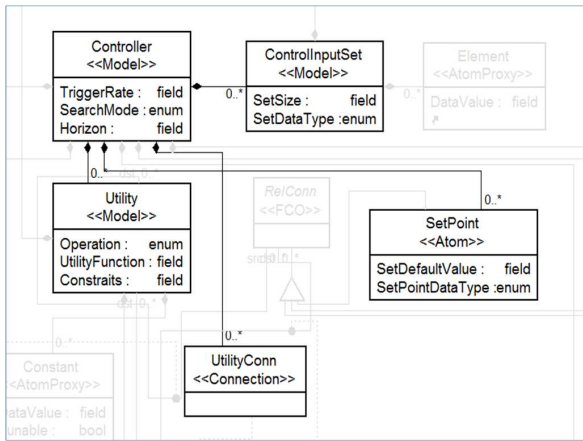


Figure 3: LLC meta-model

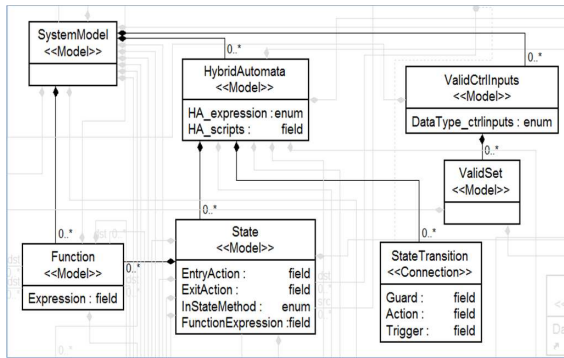


Figure 4: meta-model of the System Dynamics

controller, which has an attribute *Horizon* specifying the prediction horizon of the LLC. It contains Utility, ControllInputSet, and SetPoint models. The Utility has three important attributes: *Constrains* includes the constraints the system need to follow, *UtilityFunction* is to write the utility function, and *Operation* decides whether to “minimize” or “maximize” the utility function. The ControllInputSet contains all the available control inputs for the system. SetPoint is the target value that the automatic control system aims to reach. ControllInput, SystemState and SetPoint can be sent to the Utility by UtilityConn. Users can then use the LLC by setting the above values of the models without knowing the implementation details.

3.4 System Dynamics Model

The system dynamics specifies the behavioral characteristics of a computation system. The ACME has three types of models for the system dynamics: SystemModel, PhysicalSystem_{sim} and PhysicalSystem. In SystemModel and PhysicalSystem_{sim}, the behavioral characteristics are expressed by hybrid automata or mathematical functions, through which system states are updated. The general forms of HybridAutomata notation and Function notation are defined in the meta-model. In PhysicalSystem, the behavioral characteristics are the physical system states measured by the Sensor models.

The key models of the SystemModel as shown in Fig. 4

```

// Insert application specific code here
AfxMessageBox ("Starting..");
If (currentFCO)
{
    MON::Model cppMeta = currentFCO->getFCOMeta();
    string kindNm = cppMeta.name();
    if ( kindNm == "System")
    {
        //Traverse class, responsible for traversing the model
        Traversal tr;

        //Set the root folder of the model to the project parameter
        tr.SetParams (Model (currentFCO));
        tr.TraverseAll ();
    }
}
void Traversal::TraverseAll ()
{
    generateTreeCode ();
    generateEstimator ();
    PrintStructures ();
    generateHACode ();
    generateMainCode ();
}
    
```

Figure 5: Navigating the object network

are HybridAutomata, Function, and ValidCtrlInputs. The HybridAutomata has State models, including one InitialState in each HybridAutomata, and StateTransition connections between them. State has attributes *EntryAction*, *ExitAction*, and *FunctionExpression*; Transition has attributes *Action*, *Trigger*, and *Guard*. The transitions can be addressed in the attribute *HA_scripts* of, or modeled inside the HybridAutomata by choosing from the *HA_expression* attribute, “Using scripts” or “Embed HA inside”. The HybridAutomata model also has two aspects: **FSMAAspect** and **DataFlowAspect**. In the **FSMAAspect** state transitions are visible, while the **DataFlowAspect** demonstrates how data flow into States. The Function model has an *Expression* attribute that captures mathematical relations. The ValidCtrlInputs checks the validity of the control inputs sent by the controller corresponding to current system states. For example, if there are two States: Idle and Active, the ValidCtrlInputs should also have two ValidSets like IdleSet and ActiveSet correspondingly. Assume that the system is in the Idle State, then if a control input is not in the IdleSet, it is considered invalid; otherwise it is valid.

PhysicalSystem_{sim} model is used to simulate the behaviors of physical systems. Similar to the SystemModel, PhysicalSystem_{sim} has HybridAutomata and Function. It also has Actuator and Sensor models corresponding to the same elements as in the real physical system.

The PhysicalSystem, working in a real-time application mode, contains Actuator and Sensor models. Sensor receives system states from, and Actuator sends control inputs selected by Controller to physical plants. Both models have two main attributes: *sampling rate* and *accuracy*. System dynamics can also be included if the system can be analytically modeled.

4. ACME INTERPRETER

Interpreters are model translators designed to work with all models created using the domain-specific GME. The translated models then can be used as sources for analyzing programs [1]. We use a framework named Builder Object Network version 2.0 (BON2) to access the ACME components and the relationships between them. The BON2 generates the basic files of the interpreter, and our work consists of writing the crucial portion of the interpreter code. First,

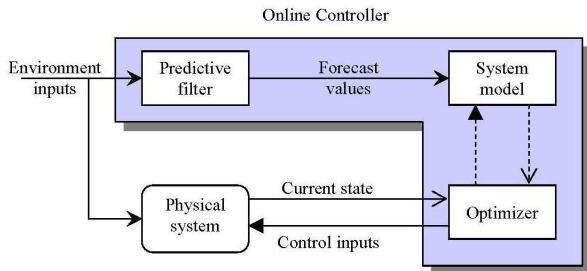


Figure 6: Structure of the limited lookahead controller

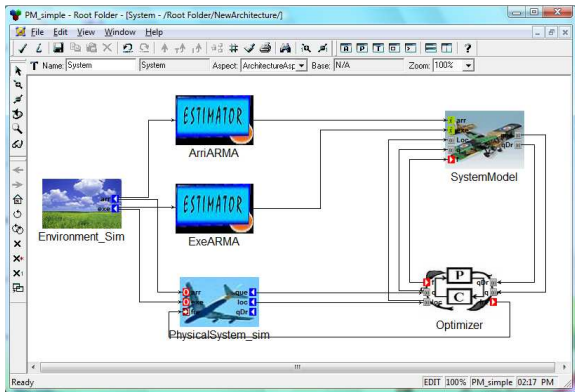


Figure 7: An ACME system implementation

the interpreter navigates the object network and traverses all the models. If a `System` exists, the traversal will start using `TraversalAll()` in the `Component::invokeEx()` function, and the `TraverseAll()` function will generate necessary files successively as in Fig. 5, when each individual component is queried by accessing its properties, attributes, meta-information, or associations. For instance, the LLC controller code identifies the `Controller` by the model property, reads the `Horizon` attribute from the `Controller`, and obtains the associated system states and control inputs. The generated scripts are ready to run for execution.

5. CASE STUDY

In this section we present a power management (PM) case study developed based on a generic limited lookahead controller framework as shown in Fig. 6. The PM uses an LLC controller to manage the power consumed by a processor under a time-varying workload. In the framework, a relevant parameter of the operating environment, workload arrival patterns, is estimated and used by the system model to forecast future system behavior over a look-ahead horizon. The controller optimizes the forecast behavior by selecting the best control inputs to apply to the system. The case study is the model integrated implementation of the PM case study as presented in [2].

The generic control framework is fully developed using the ACME tool. We build the PM application using the models generated by the ACME meta-models. Fig. 7 is a screen shot of the implemented application, which is the architecture of the system. As the case study is in the simula-

Table 1: Comparison with systems without control

	With Control		No Control		With Control/No Control
	max	average	max	average	average
Queue level	50	3.6	50	14.2	25.4 %
Dropped request	107	0.4	310	27.2	1.5 %
Frequency(Mhz)	600	342.5	400	400	85.6 %
Power cost(MJ)	360000	117306	160000	160000	73.3 %

tion mode, we use `Environment_Sim` and `PhysicalSystem_sim` models instead of `Environment` and `PhysicalSystem`. In each simulation step, two environment variables are generated in `Environment_Sim`. One is the request arrival rate obtained from a local file using the `Reader` model; the other is the execution time of the requests, set to 6.0ms in the `Generator` model. The future values of the variables are estimated by the ARMA filters and sent to the `SystemModel` to forecast two system states, queue level and dropped requests, over the `Horizon` of the `Controller Optimizer`. The queue is a buffer to store incoming requests with a limited size, so the dropped requests represents the signals dropped when the queue is full. By selecting the control input, the best CPU processing frequency, the `Optimizer` balances the forecast queue level, dropped requests, and the frequency. Finally, `PhysicalSystem_sim` updates the system states using the selected control input and new environment variables.

In each simulation step, the `Optimizer` reads the current queue level, and sends it together with all the frequencies in the `CtrlInputSet` to the `SystemModel`. The `SystemModel` will calculate all the next possible queue sizes q_i and dropped requests d_i corresponding to the i th frequency f_i . The set q_i, d_i, f_i are compared with their `SetPoint` and computed in the `Utility` model.

Each time the `SystemModel` receives new data, including current queue size and all the possible frequencies, it will check the validity of the processing frequencies for the queue size in the `ValidCtrlInputs` model. If the frequency is valid, it will be sent to the `Functions` of the `SystemModel` together with the queue size to compute the next possible queue size, which is then sent back to the `Optimizer` for further operation. Otherwise, it will be discarded.

Performance Analysis. We tested code generated by the interpreter. The performance of the power management system is evaluated using a synthetic workload file and Fig. 8 shows the results of one simulation run. The processor can operate between [200, 600] Mhz with 25 Mhz increments, and the `Horizon` of the `Optimizer` was set to 2. The request arrival rates exhibit cyclical variations characteristic of most HTTP and e-commerce workloads[4]. From the frequency responses, we can see that the controller tracks the arrival rates well. The increase in the dropped requests dues to a sustained high request arrival rate, when the controller already operates with its maximum frequency.

We compare the simulation results above with a similar system using a constant frequency 400Mhz for 10000 simulation steps, where the first 200 data are discarded considering the system adaptation. As shown in Table 1, the LLC control drops only 1.5% of the requests dropped by the constant control, while spending 73.3% of the power spent by the constant control. Moreover, if the frequency in the uncontrolled

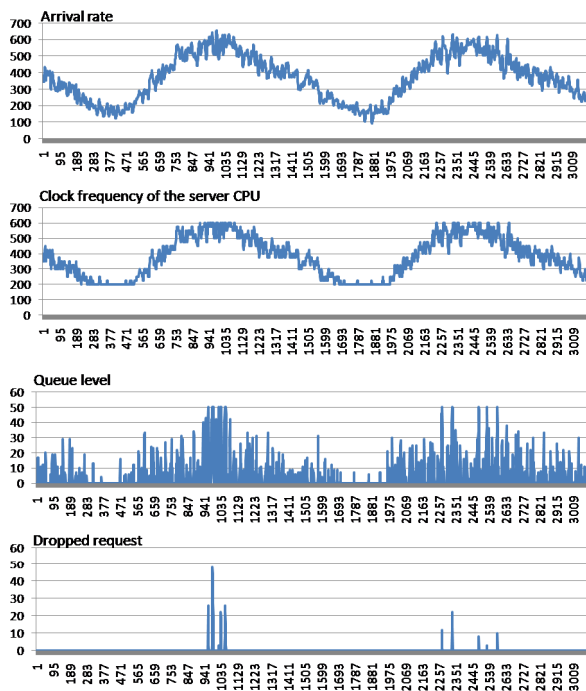


Figure 8: Simulation outcome of the power management system

model is decreased, more requests will be dropped as the processing speed of the server is slower; while an increase of the constant frequency will make the system consume more power, because the frequency of 400Mhz is already greater than the average frequency 342.5Mhz of the LLC system.

6. CONCLUSION

In this paper, we presented a model integrated framework ACME to facilitate the design of self-managed computation systems. The proposed framework can accommodate variety of model-based control strategies. Modules supporting the control structure such as estimators can be added and parameterized based on the user-defined system model and its specification. The framework provides supports of automatic synthesis of the managing controller modules based on a given system model, constraints and specification. To demonstrate the ACME framework, we developed a limited lookahead controller using the ACME framework to manage power consumption in a DVS-capable processor under a time-varying workload.

7. REFERENCES

- [1] *GME 5 Users Manual(v5.0)*, 2005. WebSite: <http://www.isis.vanderbilt.edu/Projects/gme/GMEUMan.pdf>.
- [2] S. Abdelwahed, N. Kandasamy, and S. Neema. Online control for self-management in computing systems. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium(RTAS'04)*, Toronto, Canada, May 2004.
- [3] A. Alimonda, A. Acquaviva, S. Carta, and A. Pisano. A control theoretic approach to run-time energy

optimization of pipelined processing in mpsoacs. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 876–877, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

- [4] M. F. Arlitt and C. L. Williamson. Web server workload characterization: the search for invariants. *SIGMETRICS Perform. Eval. Rev.*, 24(1):126–137, 1996.
- [5] F. Harada, T. Ushio, and Y. Nakamoto. Adaptive resource allocation control for fair qos management. *Transactions on Computers*, 56(3):344–357, March 2007.
- [6] N. Kandasamy and S. Abdelwahed. Designing self-managing distributed systems via online predictive control. Tech. report isis-03-404, Vanderbilt University, 2003.
- [7] N. Kandasamy, S. Abdelwahed, and J. Hayes. Self-optimization in computer systems via on-line control: application to power management. *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 54–61, 17-18 May 2004.
- [8] P. Kelly, A. Maulloo, and D. Tan. Rate control for communication networks: Shadow prices, proportional fairness and stability. *The Journal of the Operational Research Society*, 49(3):237–252, Mar. 1998.
- [9] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *WISP'*, Budapest, Hungary, May 24-25 2001.
- [10] C. Lu, Y. Lu, T. Abdelzaher, J. Stankovic, and S. H. Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *Transactions on Parallel and Distributed Systems*, 17(9):1014–1027, Sept. 2006.
- [11] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms*. *Real-Time Syst.*, 2006.
- [12] S. Mujumdar, N. Mahadevan, S. Neema, and S. Abdelwahed. A model-based design framework to achieve end-to-end qos management. In *ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference*, pages 176–181, New York, NY, USA, 2005. ACM.
- [13] A. Shukla, A. Ghosh, and A. Joshi. State feedback control of multilevel inverters for dstatcom applications. *Power Delivery, IEEE Transactions on*, 22(4):2409–2418, Oct. 2007.
- [14] X. Wang, Y. Chen, C. Lu, and X. Koutsoukos. On controllability and feasibility of utilization control in distributed real-time systems. *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 103–112, 4-6 July 2007.
- [15] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. Controlware: a middleware architecture for feedback control of software performance. *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 301–310, 2002.