A Model-based Simulation Approach to Error Analysis of IT Services

Long Wang, Akhil Sahai, James Pruyne
Enterprise Systems and Software Laboratory
HP Laboratories Palo Alto
HPL-2006-181
December 5, 2006*

utility computing, simulation, model, management

Utility computing environments provide on-demand IT Services to customers. Such environments are dynamic in nature and continuously adapt to changes in requirements and system state. Errors are an important category of environment state changes as such environments consist of a large number of components, and hence, are subject to errors. In this paper, we propose a framework that applies a model-based approach that enables system administrators to simulate error analyses of utility computing environments. Specifically, the information model-centric framework leverages information about existing service components and their interactions; integrates a variety of error models which are bound to individual components; captures use-case behaviors of IT services; and feeds all this information into a simulation engine. The framework also allows definitions of additional components and their interactions to provide error analysis at a finer granularity and performs service evaluation in hypothetical situations (workloads or equipment changes, use cases, error behaviors). To evaluate the framework, we performed experiments on a virtualized blade-server based environment as the target system. Results show that the framework is effective in analyzing error impacts on IT services, and hence, provides a sound foundation for designing potential error mitigation mechanisms.

* Internal Accession Date Only
A shorter version of this paper will be published in the proceedings of IM 2007, 21-25 May 2007, Munich, Germany
Approved for External Publication

A Model-based Simulation Approach to Error Analysis of IT Services

Long Wang
Center for Reliable High-Performance Comp.
University of Illinois at Urbana-Champaign,
Urbana, IL 61801
longwang@crhc.uiuc.edu

Akhil Sahai, James Pruyne
Hewlett Packard Laboratories
Palo Alto, CA 94304
{akhil.sahai, jim_pruyne}@hp.com

***Abstract*—Utility computing environments provide on-demand IT Services to customers. Such environments are dynamic in nature and continuously adapt to changes in requirements and system state. Errors are an important category of environment state changes as such environments consist of a large number of components, and hence, are subject to errors. In this paper, we propose a framework that applies a model-based approach that enables system administrators to simulate error analyses of utility computing environments. Specifically, the information model-centric framework leverages information about existing service components and their interactions; integrates a variety of error models which are bound to individual components; captures use-case behaviors of IT services; and feeds all this information into a simulation engine. The framework also allows definitions of additional components and their interactions to provide error analysis at a finer granularity and performs service evaluation in hypothetical situations (workloads or equipment changes, use cases, error behaviors). To evaluate the framework, we performed experiments on a virtualized blade-server based environment as the target system. Results show that the framework is effective in analyzing error impacts on IT services, and hence, provides a sound foundation for designing potential error mitigation mechanisms.**

Keywords-utility computing; model-based; simulation; IT service; error; error analysis; error behavior; error model

I. INTRODUCTION

Utility computing environments in the form of shared IT infrastructures and services are becoming more prevalent. These environments comprise of large range and number of components, which are inter-dependent on one another and are subject to complex interactions. Such environments are dynamic in nature and continuously experience and/or adapt to changes in requirements and system state. Specifically, utility computing environments are prone to unexpected failure behavior when the underlying components fail. As shared IT services grow popular and utility computing environments scale up, such errors are aggravated in large-scale environments and non-stop services, and impose a major threat which should be accounted for when the systems are designed.

Existing research on error analysis of component-based systems includes evaluation of system reliability/availability through system modeling or fault injection [10] [11], error mitigation of generic/specific systems [13] [12] [4], and implementation-level simulation for program debugging [14].

Unlike these works, this paper proposes a methodology to study error behavior of utility computing environments through an approach based on formal information models. Formal information model like CIM, UML are widely used in design of utility computing environments to provide flexible and general support to various IT services. Our approach leverages information about existing service components and their interactions stored in the information models to achieve the exact recreation of the utility computing environments. Besides the information models, the approach also captures use-case behaviors of IT services and introduces various error models for service components. Then all these data are fed into a simulation engine to study impacts of system errors. Moreover, additional components or interactions can be introduced into existing utility computing environments for simulation, which enables system designers to understand error behavior of proposed changes within the context of the existing service, and allows a finer-granularity of analysis on possible system errors.

In fact, the information model-based simulation approach can be applied not only to error analysis but also to study of other environment changes, e.g. hardware upgrades, software updates, addition of new features, etc. The approach is a generic methodology and can be seamlessly merged into existing design processes of utility computing environments to offer general decision support for designing IT services.

The contributions of the paper are briefly summarized as below:

- proposes a model-based simulation framework to address decision support questions for IT services;
- studies error models for a typical utility computing environment (a virtualized blade-server based environment), and designs error model representation in existing information models of the environment;
- implements a prototype of the framework for error analysis of the target utility computing environment; and
- utilizes the implemented prototype on the target environment focusing on the fail-stop error model to evaluate the effectiveness of the framework for error analysis.

Section II of the paper provides background on information model-based utility computing environments and a brief

description of our target utility computing environment. Section III presents a detailed study of error models in the target environment and the error model representation in existing information models. Section IV discusses the architecture of the model-based simulation framework. Then experiment setup and results are given in Section V and VI, respectively, before discussion of related work and conclusion.

II. TARGET COMPUTING ENVIRONMENT

The low acquisition cost of resources like PCs, memory and storage, and the high cost and complexity of resource managements make shared IT services an increasingly popular choice for enterprises. Multiple IT services are typically hosted in a utility computing environment at the same time. These services range from shared application server, web server, database utilities to the notion of a desktop utility (the target service for our case study). The environment is usually configured and managed through formal information models (e.g. UML, CIM). These information models accurately specify (a) the composition information of supported services and the computing environment, including classes of entities in the environments, instances of the entity classes, and interactions between the instances; and (b) attributes of these entity classes and instances, which are crucial for management and configuration.

The information models are stored in databases, or *model repositories*. These model repositories facilitate management in normal service delivery, and dynamic reconfiguration of IT services/ computing environments as corresponding requirements and/or policies change. Figure 1 shows the simplified architecture of our target computing environment, a commercial utility computing environment for a virtualized desktop service. The target computing environment consists of multiple physical machines (*PMs*), which are blade servers located in enclosures which in turn are hosted in racks. Each PM hosts multiple virtual machines (*VMs*). Customers log on and log off allocated virtual machines through remote desktop protocol (RDP) in the virtualized desktop service. A *connection manager* takes care of allocation of virtual machines using information stored in the model repository. Part of the information models (defined as CIM models) for the target computing environment is illustrated in Figure 2.

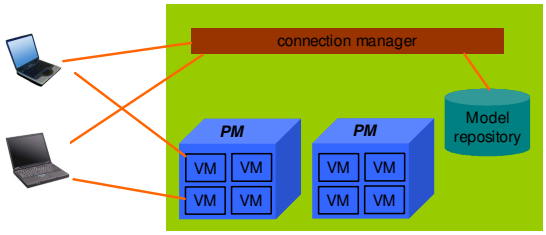

Figure 1: Simplified architecture of the target computing environment

The figure shows that, the root entity class is *CIM ManagedElement*, which is provided by the CIM information model itself. Two subclasses inherit directly from *CIM ManagedElement*: *machine* and *manager*. There are two kinds of machines, physical machine (*PM*) and virtual machine (*VM*). Two PM instances and three VM instances are shown in Figure 2 with VM1, VM2 on PM2, and VM3 on PM1. There is only one instance of *connection manager* in the environment. Interactions between instances are represented as CIM associations.

The paper focuses on error analysis of the target computing environment so that the discussion is specific and exposes in-depth sights of the proposed methodology. In fact, the methodology can be generally applied to information model-based IT service environments for addressing a variety of decision support scenarios.

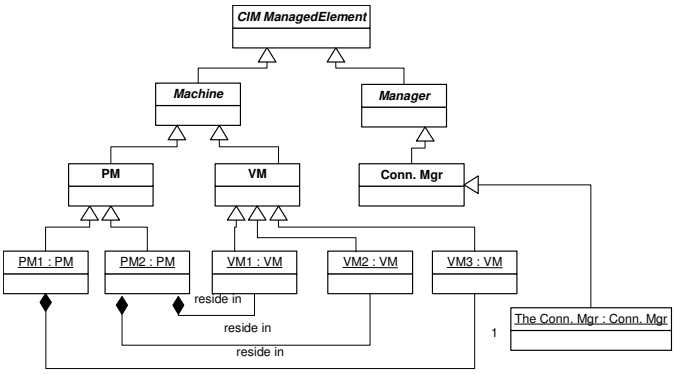

Figure 2: A sample of CIM models for the simplified architecture

III. ERROR MODEL REPRESENTATION

A utility computing environment is a complicated system consisting of a large number of components interdependent on each other. These components suffer from a variety of errors, ranging from hardware to software, from micro-scope (e.g. single bit flips) to macro-scope (e.g. entire component failure), and from benign to fatal.

Table 1 lists the error models considered in our case study for the target computing environment. A hardware component, e.g. a server node or a rack switch, may suffer from persistent defects, transient fail-stops, performance degradation or partial failures. A software component, e.g. a host OS, a VM monitor, a VM or an application, may suffer from persistent program bugs, transient fail-stops, performance degradation, race conditions, and configuration/administration errors. Every component in the environment is subject to errors, but our case study only focuses on selected components, as summarized in Table 1.

A. *Requirements for Error Model Representation*

Several requirements need to be addressed by the error model representation in the proposed framework:

- **Error categories and hierarchy have to be represented.** As error models are integrated into existing information models of the utility computing environment, the information about error types and propagations needs to be represented in form of the information model itself, i.e. errors are represented as classes and instances in the information model.
- **Error models have to be integrated without modifying models for existing components.** In design of the computing environment, components are well represented in existing information models, which include component attributes and interactions. The existing information models are complicated for real-world environments, and

are used for online management of IT services. So it is highly preferred not to make modifications to these models. For example, one possible way is to represent an error occurrence to a component as an interaction (association in CIM model) between the victim component and the error instance.

- **Component-specific error behaviors have to be captured.** Component behaviors upon the same kind of error may be different because entity instances, even in the same entity class, have their own runtime status. So the error model representation needs to capture error behaviors which are specific to individual environment components. Moreover, the error model representation should allow for specification of sophisticated error behaviors, e.g. those in performance degradation, partial failure, race condition, etc.
- **Error propagations have to be handled.** How errors propagate should be specified for each component interaction. For example, if a physical machine crashes (a fail-stop error), then all the entity instances on the physical machine (host OS, VMM, VMs, guest OSs, applications) fail with it, and in this case the fail-stop error propagates from the physical machine to these components through the "reside in" interaction illustrated in Figure 2.

The following subsection describes how the error model representation in the framework is designed to address these requirements.

B. Error Model Representation

The error models listed in Table 1 are defined as entity classes and instances in the information model, as illustrated in Figure 3 (based on the sample architecture depicted in Figure 2)[1]. An *error* is a subclass of the more general class of *change*. The *error* class has two subclasses, *hardware error* and *software error*, and each of them is partitioned into categories representing different types of error models.

Each error class has multiple instances and each error instance captures specific error specification and error behavior of an entity class of environment components. For example, usually physical machines in the utility computing environment are homogeneous and have similar error characteristics like MTTF (mean-time-to-failure). Then the error characteristics are kept as attributes of the error instance which corresponds to all fail-stop errors of physical machine (demonstrated as *HFS1* in Figure 3). The error instance also contains, as an attribute, the error behavior of the failed physical machine when the machine suffers from a fail-stop error. The error behavior is specified as a block of action statements that are interpreted during simulation process. *HFS1* only deals with fail-stop error of physical machine, and other kinds of errors for network wire, rack switch, host OS, connection manager, etc., are characterized by other error instances in the error model representation.

There are two kinds of interactions between an error instance and its corresponding environment components: *error specification* and *error occurrence*, illustrated as solid lines and dash arrows in Figure 3, respectively. An *error specification* interaction is established if an error instance characterizes the error specification and error behavior of an entity instance for the specific error model associated with the error instance. As a result, an error instance usually sets up *error specification* interactions with all entity instances of the corresponding entity class because instances of the same class usually have similar error specification and behavior. An *error occurrence* interaction is established only when an error of the specific error model happens to an entity instance. So the dash arrow from *HFS1* to *PM2* in Figure 3 means that a hardware fail-stop error occurs to the *PM2* physical machine with the error specification and behavior characterized in *HFS1*. These two interactions integrate error models into existing information models without incurring any modification to them.

Table 1: Error analysis for the target computing environment

Error Model		Involved Components	Potential error Behaviors
Hard-ware	Persistent defect	Node/blade, network wire, rack switch, storage	system/subsystem goes down every time; cannot be repaired
	Non-persistent fail-stop	Node/blade, network wire, rack switch, storage	The component fails silently; can be restarted
	Perform. degradation	Node/blade, network connection, storage	Node slowdown, storage slowdown, network congestion
	Partial failure	Rack switch, node/blade	Some connections fail; some system resources/operations unavailable
Soft-ware	Persistent error/bug	Host OS, Virtual Machine Manager(VMM), VM, guest OSs, applications, monitors, connection manager	The component goes down every time
	Non-persistent fail-stop	Same as above	The component fails silently; can be restarted;
	Perform. degrad.	Connection manager, status monitor, application	processing slowdown; delayed report; request rejection
	Race condition/timing error/deadlock	RDP server, RDP client, connection manager, system status monitor, other agents involved in use-case protocols.	The component crashes or hangs; requests denied; packets go to invalid destination
	Config/admin error	Model repository, user specification	Dependent on the component

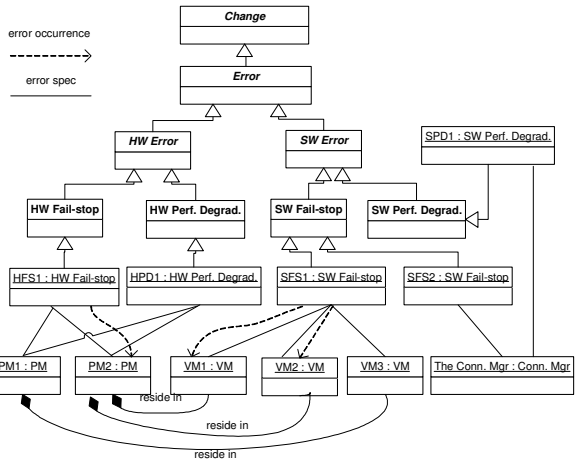

Figure 3: Representation of error models

[1] For conciseness of the figure, not all the error models in Table 1 are depicted in Figure 3, and only fail-stop and performance degradation are shown.

As mentioned above, an error instance is bound to all instances of an entity class in our design of error model representation, because usually instances of the same entity class have similar error characteristics, and it is not practical to specify error characteristics of every entity instance, which may be dynamically created in the runtime. However, as component-specific error behaviors are required to be captured, a mechanism for instance recognition is provided in the class level taking advantage of key attributes of entity instances. Each entity instance has unique key attribute values, or *ID*. Error specification and behavior are defined in the error instance using these IDs to designate individual components. Besides IDs, other attribute values of entity instances are also applied for defining component-specific error specification and behavior in the class level. Here is an example: upon a power outage *PM1* crashes while *PM2* does not crash because *PM2* has a battery, then the error behavior defined in *HFS1* may look like:

if PM.ID = "PM1" then crash;
if PM.ID = "PM2" then use_battery;

or more general,

for PMs with battery, use_battery;
for PM2 without battery, crash;

These kinds of definitions are called *rule-based* definitions. Rule-based definitions are not only used for error behaviors, but also used for specifying complicated use case behaviors in our simulation framework.

Error propagation is defined as part of error specification in an error instance. As different errors propagate through different interactions, the interactions through which an error can be propagated are specified in the error instance. For example, when a fail-stop error occurs to *PM2* (Figure 3), an *error occurrence* interaction is established between *HFS1* and *PM2*. Then the simulation engine finds that the "reside in" interaction is registered for error propagation in *HFS1*, and *VM1* and *VM2* are residing in *PM2* through the "reside in" interactions. As a result, *error occurrence* interactions are established between *SFS1* and *VM1/VM2*. (The *SFS1* is located according to the error propagation information defined in *HFS1* and the entity class information of *VM1/VM2*.) Error propagation continues until there is no interaction found for victim components to propagate the specific error.

IV. Model-based Simulation Framework

The basic idea of the model-based simulation framework is to employ event-driven simulation to mimic environment behavior under policy/state changes, or errors in our focused case study.

Each entity instance in the target computing environment is represented as a component in simulation. Existing information about entity classes and instances in model repositories, including component attributes and component interactions, is leveraged in simulation. Furthermore, a concept of *state* is introduced to all components for event-driven simulation. Upon receiving an event, a component transitions its state, updates its attributes, and/or establishes/removes interactions with other components. Manipulation of component attributes during simulation mimics real-world execution, and introduced component states allow for statistical performance measurements (e.g. the probability of a component staying in the "failed" state actually represents the unavailability of the component).

Figure 4 illustrates the architecture of the proposed model-based simulation framework. A simulation engine reads in input information from outside, performs simulation experiments, and generates experiment results for analysis. There are four kinds of input information: (a) information models of entity classes and instances (attributes and interactions) stored in model repositories (e.g. CIM model database), including error model representation discussed in Section III; (b) definitions of additional entity classes and instances which are not present in current information models; (c) behaviors of entity instances in target use-cases; (d) experiment setup and parameters (e.g. parameters of involved stochastic distributions, synthetic workload, number of physical machines in a scalability study).

There are a large number of researches exploiting simulation to study complicated systems/protocols. Our simulation framework is distinct from these works in the following aspects:

1. By employing real-world data from information models present in the real computing environment, our model-based simulation framework provides more accurate results for the target computing environment compared to a best-effort recreation of the environment, or a different hypothetical environment.

2. More sophisticated error models can be simulated in the framework because the designed error model representation permits complicated error behavior as well as error propagations through complex interactions.

3. Definitions of additional classes and instances enable analysis of the computing environment behavior, as part of the design process, for projected changes or components (i.e. errors in our case study) that are not yet present in current models of the environment. For example, network topology is not in existing design of the target computing environment. When network error, or performance impact of network topology, is considered, information on network topology can be supplied for simulation as additional classes and instances. These additional classes/instances are only used for analysis of the environment and do not bring complexity to design of the environment itself. Moreover, additional components and interactions also enable evaluation of IT services under the scenarios incapable or inefficient to be tested on existing testbeds or real workloads. Examples include scalability analysis, sensitivity study, various error models, and boundary/extreme cases.

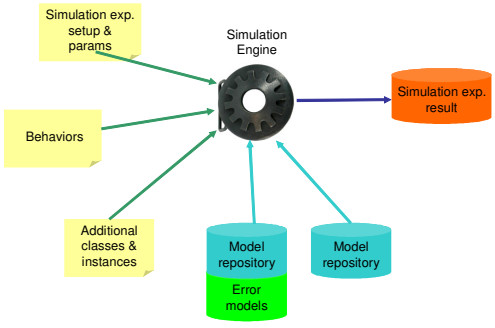

Figure 4: Architecture of the model-based simulation framework

4. Actions of entity instances can be specified in form of *action statements* which are compiled on the fly before performed in simulation. This is especially useful for simulating sophisticated error behaviors. As error specification (like MTTF) and error behavior may be dependent on current status of the victim component as well as other components, error specification and behavior are defined as action statements which are compiled on the fly to provide the required flexibility. Moreover, combination of rule-based action definition (discussed in Section III) and action statement makes the proposed approach a powerful tool to simulate sophisticated use-case scenarios and error behaviors.

Now we give more details on the architectural parts of the model-based simulation framework.

A. Simulation Engine

The simulation engine performs normal event-driven simulations. An event is a message with a target component and an event ID. An event may have or have not parameters with it. Each event is scheduled with a predefined stochastic distribution to trigger actions of the target component at a particular occasion. The stochastic distribution characterizes the semantic of the event. For example, exponential distribution is used to characterize the arrival of customer requests for logging on a virtualized desktop.

Actions taken by components include event generation, state transition, attribute update, interaction establishment/ removal, and component creation/destroy. A component may take multiple actions upon receiving an event, and these multiple actions are called an *action block*.

Though normal event-driven simulation is performed, the simulation engine is actually information-model centric. There is an action matrix specified in each entity class (and error class), which defines how instances of the entity class act upon receiving particular events. Every entity instance is in a state at any time. Let $S = \{s_1, s_2, \ldots\}$ denote the set of states the entity instance may be in; $E = \{e_1, e_2, \ldots\}$ denote the set of all events with different event IDs (so e_i and e_j have different event IDs if $i \neq j$); and A denote the set of all action blocks. Then the action matrix can be expressed as a mapping function $f: S \times E \rightarrow A$

$$f(s_i, e_j) = a_{ij},$$

where $a_{ij} \in A$ is the action block performed by the entity instance when it is in the state s_i and receives the event e_j.

However, actions taken by components may depend on the status of the component and/or other components (i.e. attribute values, component states, interactions, etc.). For example, the event of "user logon" can only trigger actions of virtual machines which are allocated by the connection manager (i.e an "allocated" attribute of the virtual machine is set as "true"). This status dependency is captured by imposing a condition before an action block. Actually, this is exactly the design of rule-based actions. A condition is a first-order predicate concerning attribute values, component states/interactions, and event parameters. Let C denote the set of all conditions, CA denote the set of all <*condition, action block*> pairs (i.e. $CA = C \times A$), then the action matrix is refined to be a mapping function $fx: S \times E \rightarrow Ax$

$$fx(s_i, e_j) = ax_{ij},$$

where $ax_{ij} \in Ax = \{ ax \mid ax \subseteq CA \}$, i.e. ax_{ij} is the set of <*condition, action block*> pairs that are performed by the entity instance when it is in the state s_i and receives the event e_j. A <*condition, action block*> pair is executed in this procedure: if the condition is satisfied, the corresponding action block is executed; otherwise, the action block is not executed. All the <*condition, action block*> pairs in ax_{ij} are executed one after another.

Though the formal specification of the action matrix above appears complex, actually the structure of the action matrix, as illustrated in Figure 5, is simple and easy to understand.

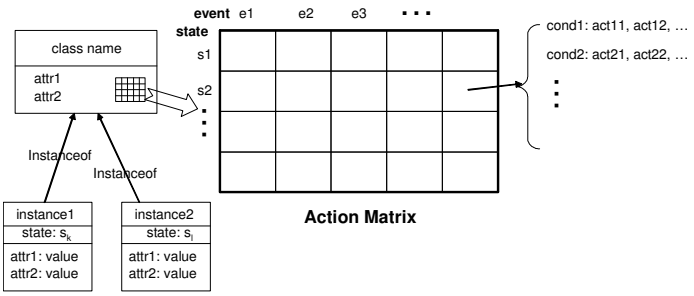

Figure 5: Structure of the action matrix in an entity class

The action matrix is placed in the entity class (and error class) instead of entity instance because it is not convenient or practical to specify the action matrix for every instance, especially in the cases when instances are created/ destoryed dynamically, or there are a large number of instances of the same entity class (e.g. in scalability study). Note that component-specific actions are rule-based, and the involved components are pinpointed by means of key attribute values of entity instances. Section III has more discussion on this issue.

B. Simulation Input Language

Figure 4 shows that, besides information models (including introduced error models) which are read into the simulation engine through a model reader provided by the information model (usually a database client), the other information, including definitions of additional classes/instances/interactions, specifications of use-case behaviors, and experiment setup/parameters, also needs to be read into the simulation engine. An XML-similar markup language is designed for this purpose, and a parser is created to compile the files written in the simulation input language. The parser can be invoked on the fly during simulation to support action statements.

Behaviors of use-cases are captured as event actions of involved entity instances. All event actions of entity instances are inputted into the simulation engine as action matrixes defined in corresponding entity classes (an action matrix can be directly specified using the simulation input language).

C. Simulation Results

There are two kinds of results out of simulation experiments.

- If the computing environment is not stable in working after an error occurrence and falls in an absorbing state, the experiment result is a recorded sequence of triggered events (and the corresponding actions), which are useful

for root-cause analysis. An example is a failure of the connection manager without any recovery mechanisms. In this case the entire computing environment stops working and no statistic performance measurements can be obtained.

- If the computing environment remains stable in working after an error occurrence, statistic performance measurements can be collected. For example, response time, throughput, availability, and utilization can be measured when servers in the computing environment undergo regular maintenance. In this case the behavior impact is performance degradation of a few percentages. (Of course, the result of event sequence is also available in this case.)

V. EXPERIMENT SETUP

This section describes setup of experiments for evaluating the effectiveness of the model-based simulation framework in addressing error-related decision support problems on the target computing environment.

A. Experiment Assumptions & Target Use-Cases

As described in Section II, the target computing environment is a real-world commercial utility computing environment for virtualized desktop service. There are multiple use-cases in the service, while in our experiments only two of them are targeted: user logon and user logoff.

The architecture of the computing environment depicted in Figure 1 is a simplified version, and more components are present in the environment. Here we give the composition of the environment components involved in experiments, as well as the assumptions made for these experiments:

- There is only one connection manager in the environment.
- There are several physical machines (PMs).
- There are multiple virtual machines (VMs) on each physical machine.
- There is one virtual machine agent (VM agent) residing in each virtual machine, which monitors the user logon/logoff status of the virtual machine, and reports status change to the connection manager.
- When a user is to log on a VM, a RDP (remote desktop protocol) client is launched, and the RDP client connects to the VM.
- There are multiple users who log on and log off VMs for their work. Each user requests a VM and logs on it at an exponentially distributed interval. A user may operate multiple VMs at the same time. Each user logs out of a VM after operating the VM for an exponentially distributed period of time.
- Overheads like processing time, network delay are ignored in the experiments. So actions triggered by events are finished immediately.

The target use cases of user-logon and user-logoff are illustrated in Figure 6 and Figure 7, respectively. They are presented in event-sequence semantics. We briefly discuss the two use cases here as understanding of the use cases is a necessity for analyzing experiment results.

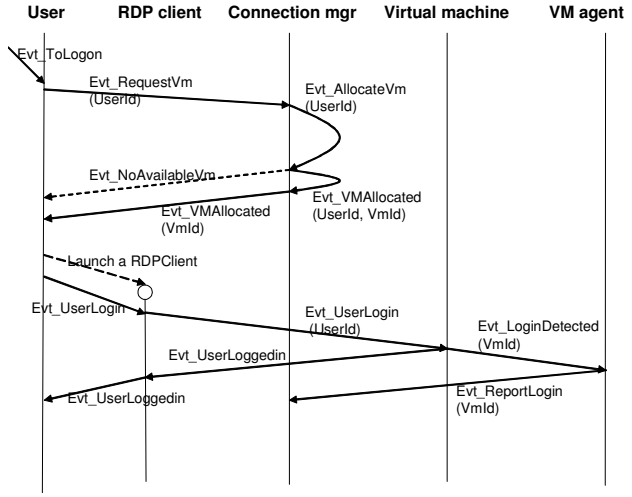

Figure 6: The use case of user logon

User logon. When a user wants to log on a VM, the user sends a request to the connection manager (*Evt_RequestVM*). On receiving such a request, the connection manager takes some time to allocate a virtual machine for the user (*Evt_AllocateVM*). If there are available VMs, the connection manager randomly picks one and sends the ID of the VM to the user (*Evt_VMAllocated*); if there is no available VM, the connection manager notifies the user, too (*Evt_NoAvailableVm*). When the user receives an allocated VM ID, he/she launches a RDP client, and logs on the virtual machine through the RDP client (*Evt_UserLogin*). After the user successfully logs on the VM, the VM notifies the user of the success (*Evt_UserLoggedin*), and then the user is able to operate the VM now. At the same time, the VM agent on the VM detects the user logon (*Evt_LoginDetected*), and reports the status change to the connection manager (*Evt_ReportLogin*).

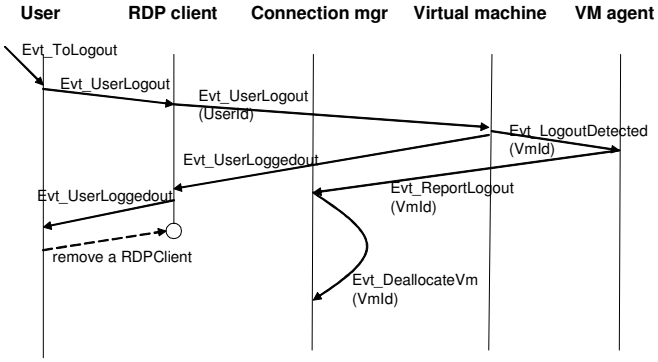

Figure 7: The use case of user logoff

User logoff. After the user operates the VM for some time, the user logs off the VM (*Evt_UserLogout*) through the corresponding RDP client. When the logoff is successful, the VM notifies the user of the success (*Evt_UserLoggedout*), and the user then terminates the RDP client. At the same time, the VM agent on the VM detects the user logoff (*Evt_LogoutDetected*), and reports the status change to the connection manager (*Evt_ReportLogout*), which then reclaims the VM for future use (*Evt_DeallocateVm*).

B. Focused Error Models

As discussed in Section III, the model-based simulation approach is able to simulate sophisticated error models such as performance degradation, partial failure, etc., by means of the error model representation (error specification and error behavior), and this feature is a major strength of the proposed framework. But in the first stage of the project we aim at the error model of fail-stop in our experiments, with the objective to evaluate the correctness of the framework design as well as the effectiveness of the framework for error analysis by resting on a simple error model and the first version of the prototype implementation. More sophisticated error models are to be analyzed on the refined implementation of the framework prototype in the near future.

The focused error model in the experiments is transient fail-stop error to any component (hardware or software) in the environment. No error detection or mitigation mechanisms are considered in the experiments.

C. Prototype Implementation

The first-version prototype of the model-based simulation framework is implemented using JDK 1.5.0 on a real utility computing environment, which was designed with CIM information model. The model repositories in the environment are stored in MySQL 4.1. A simulation tool package, DESMO-J 2.1.1 [15] [16], is employed for basic event scheduling and a library of common stochastic distributions.

VI. RESULTS

We conducted experiments for single-error scenarios, multi-error scenarios, and error-free scalability studies. The results of these experiments are presented below.

A. Single-Error Scenarios

The experiment setup for single-error scenarios is summarized below:

- There are 5 users.
- Each user requests a VM and logs on it at an exponentially distributed interval with the mean value of 4 hours.
- Each user logs off a VM after operating it for an exponentially distributed period of time with the mean value of 2 hours.
- The information of PMs and VMs is directly obtained from the model repository of the real environment. During the experiments the environment testbed had 3 PMs. One PM had 7 VMs while the other two PMs had no VMs.
- Each experiment lasts 10 days of simulated time.
- During an experiment a single fail-stop error is injected into a randomly picked instance of a particular entity class (physical machine, virtual machine, RDP client, connection manager, or VM agent) with an MTTF of 2 days. This means that, after an exponentially distributed period of simulated time the error is injected, and the mean value of the time period is 2 days. So there are cases when the time periods are longer than the experiment duration (10 days) and errors are not injected. Error injection is performed by establishing an *error occurrence* interaction between the error instance and the selected component. Then the corresponding error behavior is triggered and the error is propagated if possible.

The goal of the single-error experiments is to study the impacts of fail-stop errors which occur to individual components, or more specifically, to answer the question "what happens after a fail-stop error occurs to the component". After analysis of experiment results (event sequences), the outcomes are presented in Table 2, which includes error injection numbers and error behaviors.

Table 2: Experiment results for single-error injections *

Injected entity	Injections	Error Behaviors	
Physical machine	98	59	The PM fails (the PM does not host VMs).
		39	The PM fails -> the 7 VMs fail -> the 7 VM agents fail (the PM hosts VMs). (a) A user requests a VM, a failed VM is allocated, but when the user connects the VM by an RDP client, there is no response; (b) A user was working on a failed VM, and does not receive any response from the VM.
Virtual machine	100	73	The VM fails -> the VM agent fails (the VM is not logged on). Same as (a) above.
		27	The VM fails -> the VM agent fails (the VM is logged on). Same as (b) above.
RDP client	92	92	The RDP client fails. The connected VM will not be logged off and is unavailable for future allocation.
Connection mgr	99	99	The connection manager fails. (a) A user requests a VM but receives no response from the connection manager; (b) A user logs off a VM successfully, but the VM is not deallocated, and the config DB stays inconsistent.
VM agent	100	63	The VM agent fails (the associated VM is not logged on). A user requests a VM and the associated VM is allocated. The user logs on it and logs off it successfully. But the user logoff is not detected and the VM is not deallocated for future use.
		37	The VM agent fails (the associated VM is logged on). The user logs off the associated VM successfully, but the user logoff is not detected and the VM is not deallocated for future use.

* 100 experiments were conducted for each row in the table.

100 experiments were conducted to inject errors into instances of each entity class involved in the target use cases, and not all these experiments have errors injected (e.g. 98 out of 100 experiments are error-injected for physical machines).

The question "what happens after the error" is answered in the "error behaviors" column in Table 2. The simulation results show that (the first row in Table 2), in 39 out of 98 error-injected experiments, the PM hosting 7 VMs fails and the error propagates to the VMs and VM agents on the PM. Two scenarios happen after the error (an experiment may have both occur because multiple VMs fail in one experiment):

(a) A user requests a VM, and a failed VM is allocated to him/her. But when the user tries to connect to the VM through an RDP client, the user receives no response from the VM.

(b) A user was working on a VM when the error fails the VM. Then the user does not receive any response from the VM.

In the simulation results the user receives no response from the VM and the VM appears hung. This gives us hints that a timeout mechanism needs to be set up for detecting VM failures. Actually such a mechanism is usually provided by the underlying network protocol, and can be incorporated into the simulation through definitions of additional classes and instances.

The impacts of errors occurring to VMs, RDP clients, the connection manager, and VM agents are also listed in Table 2. Though errors of different components bring about different error behaviors, all the error behaviors in Table 2 can be roughly classified into two types with regard to VM health: (i) the VM is failed and appears hung to users (*VM failure*); (ii) the VM is not failed but is not allocated for future use (*VM wasting*). Failures of PMs and VMs lead to *VM failure*, while failures of RDP clients, connection manager, and VM agents lead to *VM wasting*.

B. *Multi-Error Scenarios*

The previous subsection studies single-error scenarios and demonstrates the effectiveness of the proposed framework in addressing the question of "what happens after the error". The study provides us insights into behaviors of virtualized desktop service when a fail-stop error occurs to a component of the service. However, in reality every component is subject to errors and multiple components may fail during a period of time. This subsection studies behaviors of the target computing environment in multi-error scenarios by allowing multiple components to fail in every experiment.

The experiment setup for multi-error scenarios is summarized here:

- A hypothetical testbed of 4 PMs, with 5 VMs on each PM, is employed in the experiments. We use the hypothetical testbed instead of the real environment testbed because our real testbed has 3 PMs and only one of them hosts VMs. As conducting experiments on a one-PM testbed is not so interesting, the hypothetical testbed is used in this study.
- During an experiment each instance of the entity classes involved in the target use cases (physical machine, virtual machine, RDP client, connection manager, and VM agent) is injected with a fail-stop error with an MTTF of 2 days.
- All the other conditions are the same as those in single-error scenarios.

Our previous study shows that, errors of the considered components bring about impacts in two categories, *VM failure* and *VM wasting*. Therefore, impacts of errors of multiple components can be observed by monitoring how many VMs are failed and how many VMs are wasted at an occasion. Usually multiple errors may bring complicated error correlations. However, as simple fail-stop errors are considered in our experiments, there are only limited types of error propagations (e.g. from a PM to the VMs and VM agents on the PM), and other types of correlated errors are not considered [2].

[2] Hopefully we will have more interesting and insightful results when correlated errors are fully studied.

We conducted 1000 experiments with multiple errors. In each experiment the numbers of failed VMs, wasted VMs, and usable VMs are recorded every 2 hours. Then averages of these VM numbers in the 1000 experiments are calculated and depicted in Figure 8.

From the figure we see that, when the experiment starts, all the VMs are usable (the environment has a total of 20 VMs). As the experiment proceeds, users log on and log off VMs, and components are subject to errors with the MTTF of 2 days (48 hours). Figure 8 shows that the number of usable VMs drops quickly and the number of failed VMs increases quickly as VMs and/or PMs fail independently. The number of usable VMs drops faster than the increase of failed VMs, because some VMs are wasted. An interesting observation is that, the number of wasted VMs first increases, and then decreases after it reaches the peak. This is because more failures of VM agents, RDP clients and connection manager bring about more wasted VMs, and these wasted VMs also fail as time continues. The figure shows that, no VM can be used after 48 hours (consistent with the uniform MTTF value), however, not all the VMs are failed at that time (about 2.4 VMs are wasted), and only after 112 hours all the VMs in the environment are failed. The figure also shows that after 14 hours the number of wasted VMs reaches the peak value of 5.1.

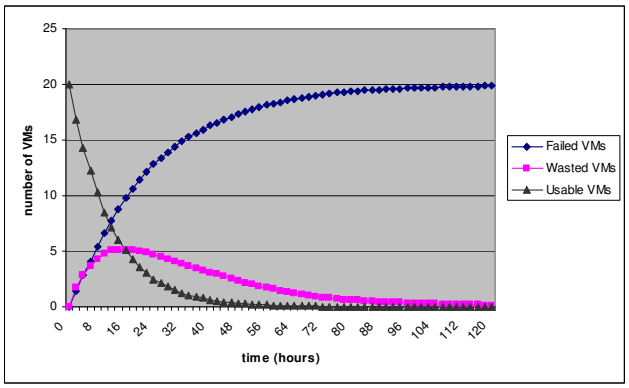

Figure 8: Evolvement of VMs in a multi-error scenario

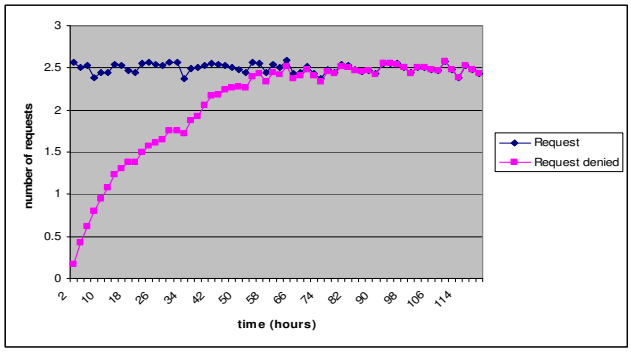

Figure 9: Servicing user-logon requests in a multi-error scenario

The performance of servicing user-logon requests is depicted in Figure 9, which illustrates how many requests are initiated in every 2 hours and how many of them are denied. Similar to Figure 8, the plotted values in Figure 9 are averages of 1000 experiments. The figure shows that, around 2.5 requests are initiated by the 5 users in every 2 hours (consistent with the experiment setup that a user requests a VM after an exponentially distributed period with the mean value of 4

hours). As there are sufficient VMs in the environment for the 5 users, few requests are denied in error-free scenarios. When components get failed as time goes along, more requests are denied. Figure 9 shows that, after 54 hours almost all the requests are denied.

With the objective to demonstrate the correctness and effectiveness of the framework design, a simple setup is exploited in our experiments, i.e. the environment components have the same error characteristics (i.e. MTTF). In reality different entity classes have different error characteristics, and the parameters of the error characteristics should be collected from the real environment to give an accurate modeling. If these features are incorporated in the experiments, the results will be very interesting and provide in-depth understanding of error-present service behaviors on the real-world target environment.

C. *Error-free Scalability Study*

The previous two subsections study the behaviors of the computing environment in both single-error and multi-error scenarios. Actually the simulation tool can be applied for studies of error-free scenarios as well, including validation of use-case designs, performance evaluation, and scalability study. Here we give a simple example demonstrating how a scalability study can be conducted through our model-based simulation framework.

As the size of the computing environment grows to a large scale, there arise challenges of effective and efficient management of the environment. Also, the number of users may easily increase and poses a major scalability problem for existing services. Since management of the environment is not involved in our target use cases, the scalability of the user numbers is focused on in this subsection.

User behavior under this study is the same as in the previous two studies. We vary the number of users from 5 to 1280 in our experiments, and then monitor how the requests from these users are serviced by the environment. For this purpose, the averages of the numbers of requests which are initiated, serviced, and declined during 2-hour intervals in each experiment are calculated (the numbers during multiple 2-hour intervals are stable in each experiment in error-free scenarios, and hence can be averaged), and these per-experiment values are averaged among multiple experiments with the same user number. Then the results, or the average numbers of requests which are initiated, serviced, and declined during 2 hours, are plotted in Figure 10 vs. different numbers of users (note that the horizontal axis has a logarithmic scale).

Figure 10 shows that, there exists a threshold of the user number such that, when the user number is less than the threshold almost no request is declined and every request is serviced (the "declined" lines lie on the horizontal axis, and the "serviced" lines coincide with the "initiated" lines); when the user number is larger than the threshold, only a fixed number of requests can be serviced and the extra requests are declined (the "declined" lines leave the horizontal axis after some point and jump high with the "initiated" lines, while the "serviced" lines depart from the "initiated" lines and remain quite stable as the user number increases). For example, when there are 20 VMs in the environment, the threshold is between 20 and 80, according to the figure.

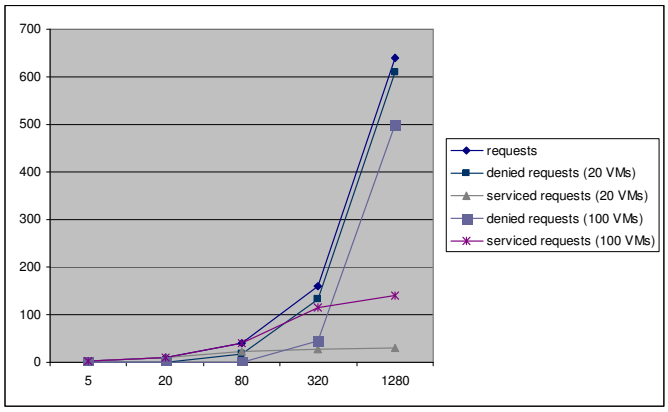

Figure 10: Servicing user-logon requests for different numbers of users

In an environment with more VMs, the threshold is increased. Figure 10 shows that, when the number of VMs in the environment increases from 20 to 100, the threshold then falls between 80 and 320.

The rationale behind the threshold is that, the computing environment has a service capacity. Let n, m, λ, and μ denote number of users, number of VMs, per-user request rate, and per-VM service rate, then the following inequation must hold for incoming workloads to be contained in the service capacity of the environment:

$$n\lambda \le m\mu$$

where $m\mu$ is the service capacity. In our experiments, $\lambda = 1/4$, $\mu = 1/2$, then the maximum user number is 40 for 20 VMs, and 200 for 100 VMs. This explains the thresholds demonstrated in Figure 10.

VII. Related Work

Reliability of IT services is a research focus nowadays in academy and industry with popularity of IT services. Traditional research work targets reliability of the specific web service. For example, Tsai et al. [1] propose a service-oriented model to evaluate reliability of web services by employing a particular testing technique, and IBM [2] applies the semantic of object transaction to web service for providing web service reliability. These models and solutions are specific to web service and can not be easily adapted to other IT services.

Utility computing and Grid computing make IT service popular and generic approaches on reliability of IT services grow prosperous. Dai et al. [3] provide a theoretical analysis of Grid service reliability by simplifying real-world Grid services into virtual-tree structures and applying graph theory and Bayesian analysis onto the structures. This is a theoretic work which ignores the complexity of real-world computing environments. Candea et al. [4] propose a general methodology to improve availability of component-based systems, typically computing environments for IT services. The methodology handles only the failures that can be recovered from reboots, and the idea is to reboot only the components that need to be rebooted for recovering from such a failure. The research has the limit that it only considers a specific category of errors, and moreover, it emphasizes on recovery mechanism provided the overall view of fault propagation paths is known, but does not

try to propose a solution for analyzing error behavior of complicated environments.

Some related works exist in error model classification and representation. A thorough and complete classification of error models is given in [5], and a classical approach, fault tree analysis [6], is widely accepted for failure analysis of complicated system. Our approach builds the error model classification into standard information models (CIM and UML) which are popularly employed in industry, and, unlike the fault tree analysis which targets analysis of abstract system composition/specification, our approach closely binds itself to real-world computing environments and provides concrete service behaviors in presence of a variety of errors.

Dependability modeling and simulation is another related research area and has lots of interesting works. Stochastic models, e.g. Markov chain, Petri Net, SAN, are widely used. [7] models a satellite network for dependability evaluation, and [8] designs a stochastic model for scalability study of deploying coordinated checkpointing protocols in large-scale systems, with emphasis on system dependability and performance. These models are either best-effort recreation of computing environments or simply hypothetical ones, and can not expose concrete service behaviors in various error scenarios.

There is also work done for evaluating reliability/ availability of real systems/environments. Fault injection is popularly used for this purpose. NFTAPE [9] is a dedicated software toolset for injecting errors into real systems and assessing system dependability, and has been applied in analyzing dependability of a variety of libraries, platforms, and system composites. For example, [10] studies error behaviors of Ensemble, a reliable group communication library, and [11] characterizes Linux kernel behaviors under errors, by deploying NFTAPE toolset.

VIII. Conclusion & Future Work

IT services as a novel computing model for E-business are prevalent nowadays and commercial utility computing environments are emerging to provide support to this computing model. Reliability is a crucial property of such computing environments for service delivery. In this paper, a model-based simulation framework is designed to analyze behaviors of these environments under different error models, ranging from simple fail-stop to sophisticated partial failure, performance degradation, and timing error. The framework combines error models with component models existing in current design of real-world utility computing environments, and hence, achieves the goal of providing a practical way to accurately evaluate error behavior of real-world computing environments. Besides error analysis on existing components in the environments, projected system designs/changes and use-case scenarios can also be tested on this framework. Experiment results demonstrate that the framework is effective for analyzing error behavior of complicated computing environments.

Current framework prototype implements only fail-stop errors. We are incorporating partial failure, performance degradation and other error models into the implementation. Moreover, error models are considered in the framework as a specific example of general environment changes. So we will apply the framework to deal with other environment changes in real computing environments, e.g. server upgrading, service specification change, service patching, etc., in the future.

References

[1] W.T. Tsai, D. Zhang, Y. Chen, H. Huang, R. Paul, and N. Liao (USA), "A Software Reliability Model for Web Services", Proceeding of Software Engineering and Applications, MIT Cambridge, Nov. 2004.

[2] Thomas Mikalsen, Isabelle Rouvellou, Stefan Tai, "Reliability of Composed Web Services : From Object Transactions to Web Transactions", white paper, IBM T.J. Watson Research Center, Oct. 2004.

[3] Yuan-Shun Dai, G. Levitin, "Reliability and performance of tree-structured grid services", IEEE Transactions on Reliability, Vol 55, Issue 2, June 2006.

[4] George Candea, James Cutler, Armando Fox, "Improving Availability with Recursive Microreboots" A Soft-State System Case Study", Performance Evaluation Journal, vol. 56, nos. 1-3, March 2004.

[5] Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C., "Basic concepts and taxonomy of dependable and secure computing", IEEE Transactions on Dependable and Secure Computing, Vol 1, Issue 1, 2004.

[6] Richard E. Barlow, "Reliability and Fault Tree Analysis", published by Society for Industrial and Applied Mathematic, 2nd Ed., 1982.

[7] E. Athanasopoulou, P. Thakker, and W. H. Sanders "Evaluating the Dependability of a LEO Satellite Network for Scientific Applications", Proceedings of the 2nd International Conference on the Quantitative Evaluation of Systems (QEST), Torino, Italy, September 19-22, 2005, pp. 95-104.

[8] Long Wang, et al., "Modeling Coordinated Checkpointing for Large-Scale Supercomputers", DSN 2005.

[9] D. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. Iyer, "Dependability assessment in distributed systems with lightweight fault injectors in NFTAPE", Proc. Of the Dependable Computing for Critical Applications Conf., 1998.

[10] Claudio Basile, Long Wang, Zbigniew Kalbarczyk, and Ravi Iyer, "Group Communication Protocols under Errors," Proc. 22nd Symposium on Reliable Distributed Systems, SRDS'03, Florence, Italy, 2003.

[11] Weining Gu, Zbigniew Kalbarczyk, Ravi Iyer and Zhenyu Yang, "Characterization of Linux Kernel Behavior under Errors", DSN'03, San Francisco, CA, June 22-25, 2003.

[12] Zbigniew Kalbarczyk, Ravi Iyer, Long Wang, "Application Fault Tolerance Employing ARMOR Middleware", IEEE Internet Computing, Special Issue on Recovery-Oriented Computing, March/April 2005, pp 28-37.

[13] Long Wang, Zbigniew Kalbarczyk, Weining Gu, Ravi Iyer, "An OS-level Framework for Providing Application Aware Reliability", PRDC 2006.

[14] Xuezheng Liu, Aimin Pan, Wei Lin, Zheng Zhang, "Using model checker and replay facility to debug complex distributed system", poster session, Proceedings of the twentieth ACM symposium on Operating systems principles, Brighton, United Kingdom, 2005.

[15] B. Page, E. Neufeld. "Extending an object oriented Discrete Event Simulation Framework in Java for Habour Logistics", International Workshop on Habour, Maritime & Multimodal Logistics Modelling and Simulation – HMS 2003, Riga, Latvia, Sept. 2003, pp. 79-85.

[16] B. Page, W. Kreutzer. "The Java Simulation Handbook. Simulating Discrete Event Systems with UML and Java", Shaker Publ., Aachen, Germany 2005.