Model-Driven Automated Error Recovery in Cloud Computing

Yu Sun\textsuperscript{1}, Jules White\textsuperscript{4}, Jeff Gray\textsuperscript{3}, Aniruddha Gokhale\textsuperscript{2}

\textsuperscript{1}Dept. of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294, USA
yusun @ cis.uab.edu

\textsuperscript{2}Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN 37203, USA
gokhale @ dre.vanderbilt.edu

\textsuperscript{3}Department of Computer Science
University of Alabama
Tuscaloosa, AL 35487, USA
gray @ cs.ua.edu

\textsuperscript{4}Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, VA 24061, USA
jules.white @ gmail.com

KEYWORDS
Model-Driven Engineering, Cloud Computing, Error Recovery

INTRODUCTION

With the increasing complexity of software and systems, domain analysis and modeling are becoming more important for software development and system applications. Applying domain-specific modeling languages and transformation engines is an effective approach to address platform complexity and the inability of third-generation languages to express domain concepts clearly (Schmidt, 2006). Building correct models for a specific domain can often simplify many complex tasks, particularly for distributed applications based on cloud computing (Hayes, 2008) that offer several opportunities for customization and variability.

Cloud computing shifts the computation from local, individual devices to distributed, virtual, and scalable resources, thereby enabling end-users to utilize the computation, storage, and other application resources (which forms the “cloud”) on-demand (Hayes, 2008). Amazon EC2 (Elastic
Compute Cloud) (http://aws.amazon.com/ec2/, 2009) is an example cloud computing platform that allows users to deploy different customized applications in the cloud. A user can create, execute, and terminate the application instances as needed, and pay for the cost of time and storage that the active instances use based on a utility cost model (Rappa, 2004).
In the cloud computing paradigm, the large number of running nodes increases the number of potential points of failure and the complexity of recovering from error states. For instance, if an application terminates unexpectedly, it is necessary to search quickly through the large number of running nodes to locate the problematic nodes and states. Moreover, to avoid costly downtime, administrators must quickly remedy the problematic node states to avoid further spread of errors.

Just like standard enterprise applications, cloud computing applications can suffer from a wide range of problems stemming from hardware failure to operator error (Oppenheimer et al., 2003). For example, Amazon EC2 provides limited guarantees about availability or reliability of hardware or VM instances. Operators must be prepared to re-launch VM instances when failures occur, transfer critical data to newly provisioned VM images, start critical services on new VM instances, join new nodes to virtual LANs or security contexts, or update load balancers and elastic IP addresses to reference newly provisioned infrastructure.

Although Amazon EC2 provides a user-friendly and simple interface to manage and control the application instances (Figure 1a), administrators must still be experienced with the administrative commands, the configuration of each application, as well as some domain knowledge about each running instance. Administrators must therefore be highly trained to effectively and efficiently handle error detection and error recovery. The complexity of managing a large cloud of nodes can increase maintenance costs, especially when personnel are replaced due to turnover or downsizing.

Even with experienced administrators, the process of error recovery involves the following challenges:

- It is hard to locate errors accurately with a large number of running application instances.
- It may take too much time to detect and locate an error, causing a long period of service termination or further error propagation.
- Error recovery becomes a time-consuming and error-prone task when it involves multiple and/or complex modification actions.
Without a logging function, it is hard to track past recovery actions, thereby leading to a potentially unreliable recovery.

Key error recovery knowledge may become concentrated in a few individuals, which becomes problematic when they leave the organization.

The main reason for these challenges is the lack of automated mechanisms to aid error recovery. Relying on manual actions to handle each error does not scale as the number of node instances increases.

This chapter presents a new automated approach to error recovery in cloud computing based on high-level graphical models of distributed application state to present the application status to an administrator. When administrators identify and correct errors in the model, an inference engine is used to identify the specific state pattern in the model to which they were reacting. Moreover, the administrator’s recovery actions are recorded. Subsequently, the model is automatically monitored for the previously identified error state and when it is encountered, the recorded recovery actions are automatically replayed or presented to the administrator as a potential course of action.

In this approach, a domain-specific modeling language (DSML) is first developed to define the running status of a specific cloud application. A DSML is usually a graphical domain-specific language to model the various aspects of a system (Gray et al., 2007). It tends to support higher level abstractions than general-purpose languages, resulting in less effort to specify a system. A DSML can be designed to construct a runtime model that serves as a graphical monitoring interface to reflect the running nodes and states of an application. Whenever errors appear in the cloud, they are also reflected in the model (i.e., models are relevant at runtime). A causal connection is established such that correcting errors in the runtime model triggers the same corresponding changes in the cloud. Because models are a high-level abstraction of the application instances, administering changes by editing the models (Figure 1b) is easier and more efficient than using the traditional command-line interface (Figure 1a). Moreover, changes to high-level modeling abstractions can trigger the automatic issuance of multiple low-level administrative changes in the cloud.

Recovering from errors by modifying runtime models is still a manual procedure that has similar drawbacks with respect to scalability, productivity, knowledge centralization, and repeatability. Instead of correcting erroneous model states manually for each occurrence of a fault, this chapter focuses on “Recovery by Demonstration,” which is derived from the idea of Model Transformation by Demonstration (MTBD) (Sun et al., 2009). Recovery by demonstration is a process whereby an administrator manually specifies a series of recovery actions for an error state and then the automation infrastructure automatically replays the same recovery actions when an identical error state is encountered in the future. Recovery by Demonstration is facilitated by the use of domain-specific models and model transformations. Specifically, the first time that an error
appears in a cloud application, a user must manually discover the problematic model elements associated with the error and manually demonstrate the recovery process by modifying the model to remove the error state.

This chapter describes an Eclipse plug-in called MT-Scribe, which integrates with a domain-specific modeling tool that can record all the recovery operations performed and then infer the error state from patterns in the model. After the first occurrence of the error, the model’s state is continuously checked for the error. Whenever the error pattern is matched, the recorded recovery operations are replayed automatically to correct the error. To showcase Recovery by Demonstration, this chapter presents two case studies involving the detection and recovery of errors in a 3-tiered Enterprise Java Beans (EJB) application that is executed within the Amazon EC2 platform. By recovering from some common errors, the case study demonstrates the potential for dynamic cloud application reconfiguration through Model-Driven Engineering (MDE) (Schmidt, 2006). The goal of our approach is to manually correct once and then automatically recover anytime and anywhere from the same error.

The rest of this chapter is organized as follows. We first provide background information on cloud computing, Amazon EC2, and the error detection / recovery problems in this field, followed by explaining how to build a runtime model for the running applications in the cloud computing server and how to establish a causal relationship between the model and application. Then, model transformation by demonstration is introduced. Through two case studies, we will illustrate how to use this technique to simplify the error detection / recovery processes. Related works and some lessons learned, as well as the opportunities for future work, are summarized after the case studies. Finally, we offer concluding remarks.

**ERROR RECOVERY IN CLOUD COMPUTING**

Cloud computing allows businesses to build and utilize complex hardware and network configurations that would normally require a large investment in a dedicated datacenter (Buyya et al., 2009). With cloud computing services, such as Amazon EC2 (http://aws.amazon.com/ec2/, 2009), computational services are purchased on an incremental basis. For example, EC2 allows businesses to purchase computation time, software licenses, networked data storage, queuing services, and database space on an hourly or per gigabyte basis.

The underlying cloud infrastructure is based on virtualization (Armbrust et al., 2009), which allows the environment to rapidly boot multiple custom OS images on the same hardware platform while guaranteeing separation of their address spaces and security. Each instance of an OS image is booted as a separate virtual machine (VM) instance in the cloud. The most basic operation that a developer using cloud infrastructure can perform is the specification of OS images that should be booted as VM instances in the cloud.
Cloud failure recovery is normally performed at the VM level. When a failure due to misconfiguration, hardware error, or excessive load is observed, developers must manually determine which VM instances to shutdown and which new VM instances to launch. Furthermore, after the new cloud assets are launched, developers must painstakingly replicate any configuration actions (e.g., editing XML configuration files for a Java application server) that cannot be packaged into the OS image.

Most existing cloud computing recovery mechanisms are based on autoscaling (Moreno-Vozmediano et al., 2009; Fronckowiak, 2008), which is a technique whereby VM instances are continuously monitored through a programmatic framework to detect failures and new VM instances are automatically launched when failures are detected. Autoscaling has been implemented in production cloud infrastructure systems by companies, such as Amazon and RightScale.

Existing autoscaling solutions leave a number of unaddressed research challenges. These challenges stem primarily from their focus on VM-level recovery mechanisms, which cannot always handle the complex application-specific requirements of cloud applications. The shortcomings of existing autoscaling-based recovery mechanisms can be broadly categorized into the following three areas:

1. **OS images for VMs cannot always encapsulate all configuration actions that must be taken to bring a VM into the application’s architecture.** Applications may require complex startup orderings for applications, editing of configuration files, or state injection. Although many OS facilities exist for automating such actions, they require expert knowledge, extensive testing, and complex development to implement properly.

2. **VM-level monitoring cannot capture many application-specific failure conditions.** Most cloud computing autoscaling mechanisms allow developers to specify failure conditions based on VM instance availability, CPU load, or disk I/O conditions. Application-specific failure conditions, such as poor response times of specific components in the web-tier or exceptions generated from application processes, cannot be monitored. The lack of this type of application-specific fine-grained monitoring prevents developers from automatically identifying and reacting to numerous types of failures.

3. **Implementing recovery actions requires expert knowledge to perform complex analyses to identify root failure causes and design appropriate autoscaling avoidance tactics.** Although an application operator may know how to repair a failure, they may not possess the intimate understanding of autoscaling to design a future avoidance strategy.

Next, we will describe how each of these limitations of using autoscaling-based recovery mechanisms can be overcome by applying a combination of modeling and Recovery by
Demonstration.

BUILDING RUNTIME CLOUD COMPUTING MODELS

In order to address Challenges 1 and 2 mentioned in the previous section, an accurate model of the application must be constructed that can precisely capture: 1) configuration details that cannot easily be embedded into an OS image, and 2) application-specific state information needed to identify failures. Furthermore, once this model is constructed, a causal link between the model and the running cloud application must be established to synchronize the model with the running application’s state. The remainder of this section describes how the model and causal link are constructed.

![Causal Link Established via JSON Over HTTP](image)

**Figure 2. DSML Application Model and Causal Link through JSON Over HTTP**

To precisely capture the concerns of the cloud application, a DSML is constructed that is tailored specifically for the concepts and terms in the application. Because of the significant cost and development time to implement a complex graphical editor for the customized DSML, a metaprogrammable modeling environment (Balasubramanian et al., 2006), such as GME (Ledeczi et al., 2001) or GEMS (http://www.eclipse.org/gmt/gems/, 2009), is used to build the editing environment. A metaprogrammable modeling environment provides developers with a graphical environment to specify the metamodel for the DSML.

The metamodel defines the terms, notation, and syntax of the final DSML. Moreover, in a metaprogrammable modeling environment, the metamodel specifies the visualization mechanisms for the language elements. After the metamodel is constructed, the metaprogrammable modeling environment automatically generates a graphical modeling editor for the DSML. This generative capability allows developers to rapidly construct application-specific models for capturing cloud application state. Figure 3 shows the metamodel of the DSML used for an EJB cloud application case study.

After the DSML is defined, a causal link must be established between the running application in the cloud and the model. Changes to the state of the cloud application must be communicated back to the DSML modeling tool and translated into changes in the elements of a specific model. For the case study presented in the next section, JavaScript Object Notation (JSON) (http://www.json.org/, 2009) messages sent over HTTP are used to communicate state changes from the cloud application back to the modeling tool. As shown in Figure 2, the modeling tool
polls each server’s control URI via HTTP for changes in its state. Common state events, such as
the availability of a node in the cloud, can be captured using a reusable library.
Application-specific events, such as the response time of a custom Java web application, require a
custom implementation of the monitoring logic. The control URI is a web-based URI that can be
used to receive state updates from the application and post changes that should be made to the
configuration. Change messages sent in either direction are encoded as JSON messages.

To translate application state changes into corresponding changes in the modeling tool, an
application-specific semantic transformation layer (STL) is used. The STL receives incoming
unprocessed state change messages from the cloud application and translates these messages into
equivalent changes in the modeling elements of the model to which the application is causally
linked. It is possible for the STL to reside within the modeling tool or on a remote host that serves
as a proxy to translate messages into native DSML semantics before they arrive at the modeling
tool. Currently, our implementation of STL is built using the Java Jetty
(http://www.mortbay.org/jetty/, 2009) embedded HTTP server to receive JSON messages, the
Eclipse Modeling Framework (EMF) (http://www.eclipse.org/modeling/emf/, 2009) as the model
implementation platform, and customized Java transformations built into the GEMS modeling
tool to perform the semantic mapping.

In some scenarios, a one-to-one mapping can be established between the state changes in the
application and the required changes in the model. For example, each application concept can be
tagged with an identifier that is directly mapped to a single model element representing the
concept in the model. In more detailed scenarios, complex predicate logic using Prolog or other
mechanisms can be used to synchronize the state of the application with the model
(Nechypurenko et al., 2007).

Changes from the model must also be pushed back into the cloud. In each HTTP polling
operation performed by the modeling tool, change messages, such as configuration actions that
should be run, can be posted to the control URI. The STL is also responsible for translating
changes in the model into control messages that use the proper semantics of the cloud application.

Figure 3. Metamodel of the DSML for EJB Cloud Application
The runtime model provides a direct and clear representation of the status of applications running in the remote cloud computing server. With the established causal relationship between model instances and applications, it offers a convenient and efficient interface for users to monitor, control and manage cloud computing applications.

RECOVERY BY DEMONSTRATION

Based on the runtime model interface and the causal relationship between models and applications, both error detection and error recovery can be realized. By checking the model elements, connections and their attributes, it is possible to find the erroneous states, configurations, and combinations that have manifested in the remote cloud computing application. When an error is identified, users can directly edit the models to recover from the errors.

Although error detection and error recovery through a model interface is user-friendly, if all the detection and recovery work were done manually, it would still be tedious and error-prone. For instance, if a large number of remote applications existed simultaneously, being reflected as thousands of model elements, relying on administrators to find any inconsistencies in the model elements would be a significant burden. In addition, some errors need to be detected and recovered immediately to avoid catastrophic failures or costly downtime. In these cases, even for a small number of applications, administrators have to keep monitoring and checking each model element constantly. If several errors occur simultaneously, the situation is even more difficult to rectify due to identification of the problems that are the root cause of an error. Therefore, it is important to automate both error detection and error recovery processes, rather than relying on completely manual management.

Error Recovery Automation by Model Transformation

Because the error detection and recovery involve checking and editing the model instances, it is very straightforward to translate these actions into model transformations in order to create an automated process. Model transformation (Sendall & Kozaczynski, 2003) is a core technology in MDE. A model transformation receives a source model that conforms to a given metamodel as input, and produces as output another model conforming to a given metamodel. When the source and target metamodels are different (i.e., between two different domains), the transformation is called exogenous (Figure 4a); if they are identical, the transformation is called endogenous (Figure 4b) (Mens & Gorp, 2005).

A complete endogenous model transformation specification usually consists of two parts – the precondition of a transformation and the transformation actions. The precondition specifies where and when the transformation should be carried out, while the actions define how a transformation should be performed.
Error recovery in the model interface can be considered as an endogenous model transformation - the runtime model in an erroneous state is transformed to a new one in the correct state, because in this case both the source and target models conform to the same metamodel. The precondition of this transformation specifies when and where to execute an error recovery transformation. In other words, the precondition serves as the criteria of error detection. When a model instance meets the precondition, the transformation actions will be executed, which consists of all the specific error recovery operations. Therefore, an error detection / recovery process through the model interface can be realized by a model transformation, as shown in Figure 5.

The main advantage of converting the error detection and recovery problems to a model transformation is that a number of automatic model transformation approaches already exist, which can be applied to automate the error detection / recovery tasks. The most common and mature approach to implement model transformations is to use an executable model transformation language, such as ATL (Jouault et al., 2008) or C-SAW (Gray et al., 2006). Most of these languages provide a high-level of abstraction, and support a declarative mechanism to simplify the specification of transformations. Each error detection / recovery process can be expressed through a model transformation. By executing these transformations whenever a change happens in the model instances, errors can be detected immediately by matching the precondition, and can be recovered by performing the associated transformation actions.
Limitations of Model Transformation Languages

Model transformation languages provide a powerful and convenient approach to automate error detection / recovery processes. However, they are not perfect and still present some challenges to users, particularly to those who are unfamiliar with a specific model transformation language. Although declarative expressions are supported in most model transformation languages, the transformation rules are defined at the metamodel level, which requires a clear and deep understanding of the abstract syntax and semantic interrelationships between the source and target models. In some cases, certain domain concepts are hidden in the metamodel and difficult to unveil (Wimmer et al., 2007; Kappel et al., 2006). These implicit concepts make writing transformation rules challenging. Moreover, a model transformation language may not be at the proper level of abstraction for an end-user and could result in a steep learning curve.

For cloud computing server administrators, it is not necessary to be a programmer or model transformation expert. Therefore, the difficulty of specifying metamodel-level rules and the associated learning curve may prevent some administrators from using transformation languages to leverage their experience and contribute to the design and implementation of error detection / recovery processes.

Model Transformation by Demonstration

To enable general administrators without deeper knowledge of programming or model transformations to realize model transformation tasks and support error detection / recovery, a simple model transformation approach is needed. Model Transformation By Demonstration (MTBD) (Sun et al., 2009) is an innovative approach to simplify model transformation tasks. Instead of writing the transformation rules manually, MTBD enables the inference of model transformations from the demonstration of a transformation process by end-users. Figure 6 provides an overview of the MTBD approach in the context of error recovery, which consists of five main steps.

Step 1: User demonstration and operations recording. A user-recorded demonstration provides the basis for transformation pattern analysis and inference. Whenever an error appears in the runtime model, the demonstration of an error recovery process is given by directly editing a model instance (e.g., add a new model element or connection, modify the attribute of a model element) to fix the error. An event listener monitors the operations occurring in the model editor. For each operation that is captured, all the information about the operation is encapsulated into an object, similar to a Command pattern (Gamma et al., 1995). The final list of these command objects represents the sequence of operations needed to finish a model transformation (error recovery) task.
**Step 2: Infer the transformation pattern.** Based on the recorded operations, a general transformation pattern is inferred that is independent of any model transformation language. This pattern describes the precondition of a transformation (i.e., where the transformation should be performed) and the actions of a transformation (i.e., how the transformation should be realized). By analyzing the recorded operations, meta-information of model elements and connections is extracted to construct a precondition, with actions specified by an operation sequence.

![Figure 6. Error Recovery Using MTBD](image)

**Step 3: Transformation refinement.** The initial precondition inferred in Step 2 is called the weakest precondition, which only specifies the minimum number of necessary model elements needed to execute the transformation actions. However, in practice, this precondition is usually insufficient, because it is often too imprecise to determine where and when to transform a model. For instance, a transformation may only need to be applied when a specific model element attribute is set to a precise value. For example, a transformation may need only to be applied when an attribute is set to a value that indicates an exception has occurred. MTBD provides a user-friendly interface to specify the detailed restrictions on the precondition. In some other cases, the inferred transformation actions are too specific to the demonstration, and therefore not generic enough. An example is that a user deletes two elements contained in another element, intending to demonstrate deleting all elements in it. However, the initially inferred transformation actions will only delete two elements in a certain container when applied. If three or more elements existed, only two of them will be removed, not all of them. Thus, enabling generic transformation actions also requires user refinement in this step.

**Step 4: Precondition matching.** After a pattern is summarized, it can be reused and applied to any model instance. By selecting a pattern from the repository, the engine automatically traverses the model instance to search for all locations that match the precondition (i.e., meet the criteria of the error) in the selected pattern. A notification is given if no matching locations are found.
Step 5: Executing actions and correctness checking. When a matching location is found (i.e., an error is found), the transformation actions are executed to transform the current model instance (i.e., to recover the error). The precondition matching step guarantees that operations can be executed with necessary operands. However, it does not ensure that executing them will not violate the metamodel. Therefore, each action is logged and model instance correctness checking is performed after every action execution. If a certain action violates the metamodel definition, all executed actions are undone and the whole transformation is cancelled.

Based on this approach, the error recovery can be demonstrated step-by-step so that a model transformation pattern can be inferred. The inferred and refined precondition can then be used as a criterion for error detection, while the actions recover the error. A repository has been built to store all the generated patterns. When a change happens in a model instance, all the patterns in the repository will be applied one-by-one to check if certain patterns could be matched and executed, ensuring that an error could be detected and recovered in a timely manner compared with the manual detection and recovery. In the remainder of this section, we illustrate how to apply MTBD in error detection / recovery through two examples.

Error Detection and Recovery Case Studies

Scenario 1 – Restarting a failed node

Error description: The attribute IsWorking in a Node is used to represent its working status. If this attribute is set to false, it indicates the web server has failed and needs to be restarted immediately. For example, Figure 7 shows a Node (called Pet Store Web Tier 1), which controls the web server on an EJB e-commerce site for selling pets to customers. Table 1 lists all the attributes of this Node with the associated values. The IsWorking attribute indicates that the Node has failed.

Error recovery solution: Whenever a Node fails to work, it should be replaced with a new Node that has the same structure, including any configuration actions that could not be embedded into the OS image. Therefore, the solution is to replace the failed Node with a new Node, which contains exactly the same structure and configuration as the old one except the IsWorking attribute is set to true.
Table 1. Attributes of PetStore Web Tier Instance 1 (Failed Node)

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsWorking</td>
<td>False</td>
</tr>
<tr>
<td>AMI</td>
<td>ami-45e7002c</td>
</tr>
<tr>
<td>CPULoad</td>
<td>12.0</td>
</tr>
<tr>
<td>CPULoadRateOfChange</td>
<td>1.5</td>
</tr>
<tr>
<td>HeartbeatURI</td>
<td><a href="http://ps01.aws.amazon.com/hb">http://ps01.aws.amazon.com/hb</a></td>
</tr>
<tr>
<td>HostName</td>
<td><a href="http://ps01.aws.amazon.com/hb">http://ps01.aws.amazon.com/hb</a></td>
</tr>
<tr>
<td>Name</td>
<td>PetStore Web Tier Instance 1</td>
</tr>
</tbody>
</table>

To handle this simple scenario by using MTBD, we need to first demonstrate the desired transformation process by performing the replacement operations on a concrete Node. Thus, in step 1, we select the PetStore Web Tier Instance 1 Node for our demonstration. Then, we delete this Node, and add a new one. There are four NodeService elements in the original Node (i.e., Bootstrap WAR Copy from S3, Tomcat, OpenSSH, Apache) that indicate processes that must be launched and active on the Node. The same NodeService elements should also be added in the new Node. Finally, to keep the new Node and the original one consistent, we should set all the attributes of the newly added elements to be the same as those in the old ones. For instance, there are seven attributes in a Node: IsWorking, AMI (Amazon Machine Image), CPULoad, CPULoadRateofChange, HeartbeatURI, HostName, Name. We can set each of the attributes in the new Node to be the same as the one in the removed Node through the attribute editor, except for the IsWorking attribute. It is the same case for NodeService. In total, 12 attribute editing operations are needed to set the three attributes for the four new NodeService elements.

Listing 1 shows the operations performed to replace a Node. All of these operations and related information (e.g., metamodel information and location of the operations) can be recorded in sequence. Figure 8 shows a screenshot of the demonstration in process and the operations being recorded in the view.
### Listing 1. Operations Demonstrated for Scenario 1

<table>
<thead>
<tr>
<th>No.</th>
<th>Operation Recorded</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Remove PetStore Web Tier Instance 1</td>
</tr>
<tr>
<td>2</td>
<td>Add a new Node</td>
</tr>
<tr>
<td>3 – 8</td>
<td>Set the attributes of the new Node to be those in the old one (6 attributes)</td>
</tr>
<tr>
<td>9</td>
<td>Set IsWorking of the new Node to be true</td>
</tr>
<tr>
<td>10 – 13</td>
<td>Add four new NodeService</td>
</tr>
<tr>
<td>14 – 25</td>
<td>Set all the attributes of these NodeService to be those in the old ones (3 attributes)</td>
</tr>
</tbody>
</table>

In step 2, when the demonstration is finished, the basic transformation pattern is inferred. This pattern consists of the list of actions (shown in Listing 1), as well as the weakest precondition. This precondition means that in order to execute this transformation there must be sufficient model elements for the recorded operations to be replayed correctly. Since the recorded operations are based on specific model elements (e.g., Web Tier Instance 1, Tomcat), the precondition is generalized by extracting the metamodel information in each recorded operation, and renaming each element in sequence. As shown in Listing 2, the precondition for this example is that there must be a Node, which contains at least four NodeService elements.

![Demonstration by editing the model](image1.png)

![Operations being captured in the view](image2.png)

**Figure 8. Screenshot of Demonstration Process and Operation Recording View**
However, the weakest precondition is usually not sufficient to determine when and where to apply a transformation. For instance, in this scenario, we only want to replace the Node whose IsWorking attribute is false. But if the inferred precondition in step 2 is applied, all the Nodes with at least four NodeService elements will be replaced, regardless of whether they are working or not. Therefore, more specific constraints are needed to refine the inferred precondition. In this case, the precondition should be further modified to be the Node with at least four NodeService elements with the IsWorking attribute set to false. To realize this refinement, in step 3, users are asked to give more restrictions on the generated precondition. Users can specify Node.IsWorking == false through a precondition editor dialog (Figure 9). This refinement will be integrated with the original precondition and applied in the future steps.

**Listing 2. The Weakest Precondition for Scenario 1**

<table>
<thead>
<tr>
<th>Model Elements Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
</tr>
<tr>
<td>Node.NodeService1</td>
</tr>
<tr>
<td>Node.NodeService2</td>
</tr>
<tr>
<td>Node.NodeService3</td>
</tr>
<tr>
<td>Node.NodeService4</td>
</tr>
</tbody>
</table>

Another problem related to the inferred precondition problem is that not all of the Nodes contain four NodeService elements. Consider the Data Tier Instance 1 Node as an example; only three NodeService elements are contained – MySQL, Apache, PHPMyAdmin. In this case, the inferred precondition cannot be matched even if the Data Tier Instance 1 failed to work. Therefore, we need to give some feedback information on the recorded operations and indicate that some of them are generic and can be executed with a different number of available elements. From Listing 1, it can be found that operations 10 through 13 are the same remove operations repeated, and 14 – 16, 17 – 19, 20 – 22, 23 – 25 are also four sets of repeated operations. Therefore, in step 3, we can provide a feedback result that steps 9 – 12 are the same, so only one is needed, and this set can be repeated according to the number of elements available; 13 – 24 is the four same set of operations, so only one set is needed, and this set can be repeated similarly. After this step, the final operation list is shown in Listing 3. The associated precondition can be seen in Listing 4. All these lists are shown to illustrate the abstract representation of the pattern data stored, which are not visible to administrators. The administrators only need to demonstrate the process and refine the pattern in the user-friendly interfaces (e.g., the dialog shown in Figure 9), after which all the other following steps will be carried out automatically.
After refining the transformation precondition, a complete transformation pattern is generated and saved in the repository. This pattern has a precondition, which in this case can be used as the criteria of error detection, and a sequence of actions, which are the error recovery operations. From now on, whenever a change happens in the model, all the saved patterns including this one will be executed one-by-one automatically. When executing a pattern, the engine will first match the precondition in the current model instance at step 4. In other words, the current model instance will be traversed and all the locations matching the precondition will be found. In this process, a back-tracking matching algorithm is applied. More details about this algorithm can be found in (Sun et al., 2009).

**Listing 3. Final Operation List after Refinement**

<table>
<thead>
<tr>
<th>No.</th>
<th>Operation Recorded</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Remove Node</td>
</tr>
<tr>
<td>2</td>
<td>Add a new Node1</td>
</tr>
<tr>
<td>3 – 8</td>
<td>Set attributes of the Node1 to Node (6 attributes)</td>
</tr>
<tr>
<td>9</td>
<td>Set IsWorking attribute of the Node1 to true</td>
</tr>
<tr>
<td>10</td>
<td>Add a new NodeService1 in Node1 (repeat if needed)</td>
</tr>
<tr>
<td>11 – 13</td>
<td>Set attributes of NodeService1 to Node.NodeService (repeat if needed)</td>
</tr>
</tbody>
</table>

**Listing 4. Final Precondition after Refinement**

- Model Elements Needed
  - Node (Node.isWorking == false)
  - Node.NodeService1 (repeat if needed)

Then, at those matching locations, the sequence of actions in the transformation pattern will be executed. Because the possibility of violating the metamodel definition exists when executing the transformation actions, each execution will be logged. If a violation happens, all the executed actions will be undone and the transformation will be canceled in that matching position.
Scenario 2 – Overload Management

Error description: If the CPULoad of a Node exceeds 20, and CPULoadRateofChange exceeds 5, the Node is overloaded. Table 2 shows a Node in the erroneous overloaded state.

Error recovery solution: Replace the Node with two identical Nodes, and split the CPULoad equally to the two new Nodes. In other words, set the CPULoad attribute of each new Node to be half of the original Node.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsWorking</td>
<td>True</td>
</tr>
<tr>
<td>AMI</td>
<td>ami-45e7002c</td>
</tr>
<tr>
<td>CPULoad</td>
<td>22.0</td>
</tr>
<tr>
<td>CPULoadRateOfChange</td>
<td>5.5</td>
</tr>
<tr>
<td>HeartbeatURI</td>
<td><a href="http://ps01.aws.amazon.com/hb">http://ps01.aws.amazon.com/hb</a></td>
</tr>
<tr>
<td>HostName</td>
<td><a href="http://ps01.aws.amazon.com/hb">http://ps01.aws.amazon.com/hb</a></td>
</tr>
<tr>
<td>Name</td>
<td>PetStore Web Tier Instance 1</td>
</tr>
</tbody>
</table>

Table 2. Attributes of PetStore Web Tier Instance 1 (Overloaded Node)

Similarly, we first select a Node and perform the demonstration. As shown in Listing 5, after adding the two new Node elements through the same set of operations in scenario 1, the attributes are setup as usual. In order to split the original CPULoad into two equal parts, the attribute editor is applied. For example, if the original CPULoad is 25, we can set NewNode.CPULoad = 25 / 2 = 12.5 through an attribute editor dialog, which can be internally recorded as NewNode.CPULoad = PetStore Web Tier Instance 1.CPULoad / 2. The attribute editor enables users to specify the attribute computation at the instance level in a demonstration process, but infer the transformation rules at the metamodel level, so that when the value changes at the next time (e.g., 50, not 25), it can still compute the correct value. As shown in Figure 10, the attribute editing dialog enables users to access all the attributes of the current model instance, and specify a complex attribute refactoring computation. In some of the other related approaches, which will be discussed in the “Related Works” section, only concept mapping is supported without considering attribute transformation, such that splitting the value into halves cannot be realized as used in this case.
**Listing 5. Operations Demonstrated for Scenario 2**

<table>
<thead>
<tr>
<th>No.</th>
<th>Operation Recorded</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Remove PetStore Web Tier Instance 1</td>
</tr>
<tr>
<td>2</td>
<td>Add a new Node</td>
</tr>
<tr>
<td>3 – 8</td>
<td>Set the attributes of the new Node to be those in the old one (6 attributes)</td>
</tr>
<tr>
<td>9</td>
<td>Set the CPULoad attribute of the new Node to be half in the old one</td>
</tr>
<tr>
<td>10 – 13</td>
<td>Add four new NodeService</td>
</tr>
<tr>
<td>14 – 25</td>
<td>Set all the attributes of these NodeService to be those in the old ones (3 attributes)</td>
</tr>
<tr>
<td>26</td>
<td>Add a new Node</td>
</tr>
<tr>
<td>27 – 31</td>
<td>Set the attributes of the new Node to be those in the old one (6 attributes)</td>
</tr>
<tr>
<td>32</td>
<td>Set the CPULoad attribute of the new Node to be half in the old one</td>
</tr>
<tr>
<td>33 – 36</td>
<td>Add four new NodeService</td>
</tr>
<tr>
<td>37 – 48</td>
<td>Set all the attributes of these NodeService to be those in the old ones (3 attributes)</td>
</tr>
</tbody>
</table>

![Figure 10. Attribute editing dialog](image)

The original transformation pattern inferred needs to be refined as well. In this scenario, the precondition should be all the Nodes whose CPULoad is greater than 20 and CPULoadRateofChange is greater than 5. Therefore, we added one restriction on the precondition: PetStore Web Tier Instance 1.CPULoad > 20 && PetStore Web Tier Instance 1.CPULoadRateofChange > 5. In addition, adding NodeService and attribute editing operations should be marked as generic, as in the last scenario.

Executing this transformation will automatically find all of the Nodes that have too much working load, and split the load into two new Nodes. If the load in the new Node is still over the limit, it can be split again by invoking the transformation repeatedly until the values satisfy the precondition.
RELATED WORK

Error recovery plays an important role in every aspect of computer science. A number of research efforts have investigated error recovery in fields, such as compilers (Graham & Rhodes, 1973), robotics (Donald, 1989), and network applications (Carle & Biersack, 1997). Because cloud computing is a newly emerging area, the research on error recovery in this field is still limited. In (Wang et al., 2009), Wang et al. proposed a schema to ensure the correctness of data storage in cloud computing. Recovering errors in the administration of cloud computing applications have not been thoroughly investigated.

In our approach, domain-specific modeling and model transformation are the foundation of the implementation. MTBD is the core technology in this work. Its original aim is to simplify implementation of model transformation tasks, following the similar direction of Model Transformation By Example (MTBE) approaches. Balogh and Varró introduced MTBE by using inductive logic programming (Balogh & Varró, 2009; Varró & Balogh, 2007). The idea is to generate graph transformation rules from a set of user-defined mappings between the source and target model instances by applying an inductive logic engine. Similarly, Strommer and Wimmer implemented an Eclipse prototype to enable generation of ATL rules from the semantic mappings between domain models (Strommer & Wimmer, 2008; Strommer et al., 2007). Both approaches provide semi-automatic generation of model transformation rules, which need further refinement by a user. Because both approaches are based on semantic mappings, they are more appropriate in the context of exogenous model transformations between two different metamodels, and may not be as conveniently applied in endogenous model transformations. In addition, the generation of rules to transform attributes is not well-supported in most MTBE implementations, which makes MTBE insufficient to fully support error detection and recovery discussed in this chapter.

A closely related work to MTBD is called program transformation by demonstration (Robbes & Lanza, 2008). To perform a program transformation, users first manually change a concrete program example, and all the changes are recorded by the monitoring plug-in. Then, the recorded changes are generalized into a transformation. After editing and specifying the generated transformation, it can be applied to other source code locations.

Our work involves a runtime model to reflect the status and transmit the changes of the remote applications in a cloud computing server. This is actually a typical application in the field of research known as models at runtime. An increasing number of works have been conducted in this field. For instance, in (Jouault et al., 2006), a method to extract runtime models to support reverse engineering and software evolution is introduced. Chitchyan and Oldevik (Chitchyan & Oldevik, 2006) presented the challenges and solutions of maintaining a multi-dimensional separation of concerns model at runtime and dynamic introspection and adaptation of concerns in such a model. A runtime model and its evolution for Self-Adaptation in the Ambient Assisted Living Domain are illustrated in (Schneider & Becker, 2008).
LESSONS LEARNED AND FUTURE WORK

The inferred transformation pattern can be used as a valuable source for cloud application administrator training, which is an essential process in cloud computing application management. The most effective training process must contain various examples and real cases. When teaching administrators how to detect and handle errors, the inferred transformation pattern can be used to better illustrate the problems. For instance, the precondition in a transformation pattern formally specifies the conditions of an error. By executing the pattern, it shows whether a certain sample model meets the criteria of an error according to the result of pattern matching. Additionally, the transformation actions can serve as an instructional guide on how to remedy the error. By executing the transformation pattern, the final status of the model after correcting the error also can be easily observed, so that the administrator can compare the original and final state. This is especially important when a large number of complex operations are needed to fix the error. In future work, we plan to investigate the use of a transformation debugger with forward and backward stepping to help train administrators.

The recording mechanism can also be used to realize logging so that the reason for a failure can be tracked. The current operations recording mechanism can monitor and capture a wide variety of operations from the model editor. Extending the operation recording mechanism to record all operations, rather than just operations in a demonstration process, can provide a complete logging mechanism. When necessary, the sequence of changes made on any model instance can be tracked, and the administrators’ control on the servers is enhanced. This logging functionality may be particularly useful for post-mortem analysis of failures due to administrator error.

Another benefit to recording more generic actions, rather than only model editing operations (e.g., running a model transformation, and invoking certain applications), is that it can enhance the richness of error recovery solutions. For example, if two error recovery transformations are already inferred and available, they can be applied together if needed when solving another new error. For this new error, two operations are needed to demonstrate the process, invoke the first transformation and invoke the second transformation, which are much more efficient than solving this problem step-by-step. In addition, we can also integrate other system operations into the demonstration, such as restarting the monitoring system, closing the model editor, or even opening a web browser and sending an email.

Although this work is specific to the Amazon EC2 cloud computing system, it can serve as a useful framework for controlling and managing runtime models with model transformations. For other areas, the only modifications required would be additional metamodels for the new domains. The current mechanism of model / domain causal relationship can also be reused, and MTBD works for any model instances, independent of the underlying metamodels. Therefore, in a similar area where runtime monitoring and controlling is not well-supported, our approach offers a
powerful and flexible framework to realize the related tasks. Additionally, our approach is not restricted to error recovery tasks only. Several general administration tasks in cloud computing could also benefit from this framework, such as reconfiguration of the applications based on certain criteria, adding / stopping application nodes, etc. More exploration of using our approach on these general administration tasks is one of our future works.

The design of the metamodel can be either generic or specific to the individual application. In the cloud computing domain, for instance, we could build a generic metamodel which is capable of describing all kinds of applications like the one used in the case study. However, this generic metamodel can only cover similar functions among different applications, and there must be some specific aspects in certain domains that cannot be reflected through this generic metamodel. Thus, before building a metamodel, it is necessary to conduct a complete domain analysis and decide what kind of metamodel is needed. Generally, the simpler the metamodel, the easier it will be to model the application implementation.

Although MTBD greatly simplifies certain model transformation tasks, it is not as powerful as writing general-purpose model transformation specifications. Many of the functions and mechanisms that can be expressed by a general transformation language are difficult to demonstrate and infer with MTBD. For instance, precondition specification is still a weak part of our process. Constraints can be placed on the attributes of the model elements, but it is impossible to specify the desired model structure (e.g., a Node is connected with exactly two other Nodes). Enhancing the precondition specification in MTBD will be our main future work. To further improve MTBD, more complex operations should be supported, such as adding a specific number of certain elements, or functions such as min(), max().

**CONCLUSION**

This chapter describes a novel application of using MDE to support domain analysis by automating the error detection and error recovery processes in cloud computing servers. The implementation is based on a runtime model framework that reflects the real status of applications executing in remote servers and is capable of transmitting the changes made in models to the corresponding applications. This runtime model framework not only provides a more user-friendly interface for administrators to manage and control the cloud computing applications, but also makes automating error detection and recovery processes easier to realize. MTBD is an innovative method to simplify model transformations. It can infer the precondition and needed actions of a transformation from the user demonstrated operations. Because detecting and recovering errors in the runtime models can be regarded as endogenous model transformations, using MTBD to demonstrate an error recovery process and infer a generic model transformation including the precondition (i.e., the criteria of error detection) could improve the efficiency and simplicity of the work. Based on this approach, each error must be manually corrected on the first occurrence. After that, with the inferred model transformation pattern, the same error can be
automatically detected and recovered anywhere without manual intervention. From this application, it has been found that domain analysis and management could benefit from some of the model-driven technologies, and therefore could be accomplished more appropriately in the context of MDE.

ACKNOWLEDGMENT
This work is supported in part by an NSF CAREER award (CCF-0643725) and NSF CNS Core award (CNS 0915976).

REFERENCES


