Improving Reliability in Management of Cloud Computing Infrastructure by Formal Methods

by

Shinji Kikuchi

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Kunihiko Hiraishi

School of Information Science
Japan Advanced Institute of Science and Technology

December 18, 2013

Abstract

It has been reported that the most of failures occurred in information systems have been caused by human errors such as misconfigurations and improper operations. Therefore, it is the utmost importance to prevent them. Among these human errors, we concentrate on the misconfigurations and improper operations caused by overlooking the constraints to be kept in the systems management and operations. By preventing the overlooking, we can reduce the occurrence of failures and improve the reliability of information systems. As for the constraints, we consider the constraints regarding (1) the operation executions such as "during the live migration operation, the memory size of virtual machines should not be over the capacity of the physical server on which these virtual machines are running" and (2) system configurations such as "the system should not have a single point of failure". We considered that by using analysis method such as formal methods, we will be able to determine the satisfiability of these constraints in the information systems. By feedbacking the verification results to the system administrators, we will be able to prevent the overlooking of the constraints and improve the reliability of system management.

In this thesis, we propose the following two approaches based on formal methods to improve the reliability of system management.

1. Synthesis of configuration change procedure: Using model finding approach, we automatically construct the procedures for configuration changes in ICT (Information and Communication Technology) systems. By determining the processes satisfying declarative constraints defined beforehand, we can prevent the failures caused by executions of configuration change procedures in which the administrators overlook the constraint to be satisfied in the configuration change.

2. Identification of vulnerability in system configuration: Using model checking technique, we evaluate and identify the risk (e.g. single point of failure) in system structure and configuration changes. After identifying the risks in the system, we can determine how the configuration changes can give an impact on the vulnerability in system management. By this evaluation, we can avoid the execution of risky operations and prevent the occurrence of service failures.

Acknowledgments

The author wishes to express his sincere gratitude to his principal advisor Professor Kunihiko Hiraishi of Japan Advenced Institute of Science and Techology for his constant assistance and kind guidance during this work. The author would like to thank his advisor Professor Toshiaki Aoki of Japan Advanced Institute of Science and Technology for his helpful discussions and suggestions for sub research theme. The author also wishes to express his thanks to Associate Professor Kazuhiro Ogata of Japan Advanced Institute of Science and Technology for his suggestions for this work.

The author is grateful to managers of System Software Laboratories (former Cloud Computing Research Center) at FUJITSU LABORATORIES LTD. Fellow Yoshitaka Sakashita, Head of System Software Laboratories Mitsuhiro Kishimoto, Senior Expert Motomitsu Adachi and Research Manager Yasuhide Matsumoto gave him their constant encouragements and supports. His colleagues also gave him countless supports.

The author also wishes to express his gratitude to Lecturer Radu Calinescu at University of York for his many useful advices regarding formal methods.

Last but not least, the author would like to say a big thank you to his family; his parents and brother for their support, his wife for her great encouragement for the research, and his son for many innocent smiles that give him happiness and powers for pursuiting innovative research every day.

Contents

Chapter 1

Introduction

1.1 Emergence of cloud computing

Recently, the paradigm of information systems providing various services to our society has been drastically changing. In the past, it is typical to own computing resources (e.g. servers) in a company or a datacenter to provide services to users. However, the emergence of virtualization technologies enabling to instantiate several virtual machines on a physical server has opened revolutionary usages of computing resources. Data centers having large amount of computing resources lend them to users by "pay-as-you-go" manner. This transition of computing resource invoke the drastic shifting from capital expenses to operating expenses. The users of these data centers do not have to mind where their computing resources or their data are deployed in data centers. This approach has been called "Cloud computing", because in science field a large agglomeration of objects is visually described as a cloud [6].

The emergence of cloud computing enables cloud users to start their businesses with small capital expenditure without owning their own facilities. In addition, when their businesses grow (or shrink), they can rent additional resources (or cease to use resources). This kind of elasticity and convenience has attracted many users and cloud services has been prevailing. As as result, a large number of information systems and services has been converging into cloud data centers.

1.2 Complexity in management of cloud computing

Since cloud data centers have to accommodate massive amount of computing resources to serve many users, the size of cloud data centers has been becoming larger and larger. For example, it is estimated that Google owns 900,000 physical servers and Amazon owns 450,000 physical servers in their datacenter [1, 2]. In addition, while new technologies such as virtualization has enabled new functions such as dynamic resource provisioning and live migration, they complicates the physical and logical structure of cloud datacenter infrastructure. This results in the difficulties in the situation awareness and the executions of appropriate system operations and managements. As a result, serious service failures in cloud computing services has been happening every day. For example, one of the major outages happened in Amazon Web Services in 2011 was triggered by the execution of improper operations [43]. Since the cloud computing services are used by many users, the

impact of service outage is quite enormous. From this backgroud, in the operation and management of cloud computing infrastructure, the demand for the technologies which can prevent service outages caused by misconfigurations or improper operations has been increasing.

1.3 Contribution of the dissertation

The main contribution of this dissertation is to show how to improve the reliability of operation and management for complex information systems such as cloud computing infrastructure. While there can be various types of approach to improving the reliability, the most typical approaches are (1) prevent misconfigurations beforehand and (2) identifying existing misconfigurations (or undesirable settings). For these two approaches, we propose and evaluate the following two techniques utilizing formal methods.

1. **Synthesis of configuration change procedure**

   Using model finding approach, we automatically construct the procedures for configuration changes in ICT systems. By determining the processes satisfying declarative constraints defined beforehand, we prevent the failures caused by executions of improper configuration change procedures in which the administrators overlook the constraints to be satisfied in the configuration change.

2. **Identification of vulnerability in system configuration**

   Using model checking technique, we evaluate and identify the risk (e.g. single point of failure) in system structure and configuration changes. The example of the risk is a single point of failure. We determine how undesirable events (e.g. improper configuration changes and component failures) can give an impact on the vulnerability in system management. By this evaluation, we can pay extra attention the execution of risky operations and prevent the occurrence of service failures.

1.4 Structure of the dissertation

The dissertation is organized as follows.

Chapter 1 (this chapter) introduces research background, problems in management complexity of cloud computing infrastructure, contribution and structure of the dissertation. Chapter 2 presents preliminaries related to cloud computing, system management and formal methods. In Chapter 2 we define the detail of the target system and its management to be investigated. The configuration of private cloud system and the management tasks to be focused on are presented in the chapter. It also is explained that two types of analysis methods are proposed in this thesis: (1) designing proper configuration change procedures and (2) evaluating the system's robustness for improper configuration changes. Chapter 3 explains a method of configuration change procedure synthesis using model finder. This method synthesizes a procedure satisfying constraints regarding system management represented by logical formula using Alloy Analyzer. Chapter 4 explains the operational vulnerability evaluation method using model checking. To achieve this, first we define the operational vulnerability scale representing how a system is susceptible to undesirable events. Then we execute verification to determine which vulnerability level

the system is in by NuSMV model checker. Chapter 5 presents the discussion including the application of formal methods for improving the reliabilities of tasks in system management lifecycle and related work. Chapter 6 concludes the dissertation.

Chapter 2

Preliminaries

2.1 Cloud computing

Before starting the discussion of technologies for improving the reliability of cloud system management, here we first explain the overview of cloud computing. While it is sometimes said that the definition of cloud computing is diverse and ambiguous, there are some common understandings. First, it is said that the person who is the first to use the term "cloud computing" is Eric Schmidt (CEO of Google). In Search Engine Strategies Conference in 2006, he said as follows [3].

> What's interesting [now] is that there is an emergent new model, and you all are here because you are part of that new model. I don't think people have really understood how big this opportunity really is. It starts with the premise that the data services and architecture should be on servers. We call it cloud computing – they should be in a "cloud" somewhere. And that if you have the right kind of browser or the right kind of access, it doesn't matter whether you have a PC or a Mac or a mobile phone or a BlackBerry or what have you – or new devices still to be developed – you can get access to the cloud. There are a number of companies that have benefited from that. Obviously, Google, Yahoo!, eBay, Amazon come to mind. The computation and the data and so forth are in the servers.

Since then, along with the emergence of cloud services such as Amazon Web Services [4] and Google App Engine [5], the term "cloud computing" has been becoming popular and widely used. The idea of using various services located beyond network from local devices can be represented by Figure 2.1.

NIST (National Institute of Standards and Technology) summarized their definition of cloud computing as follows [7].

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

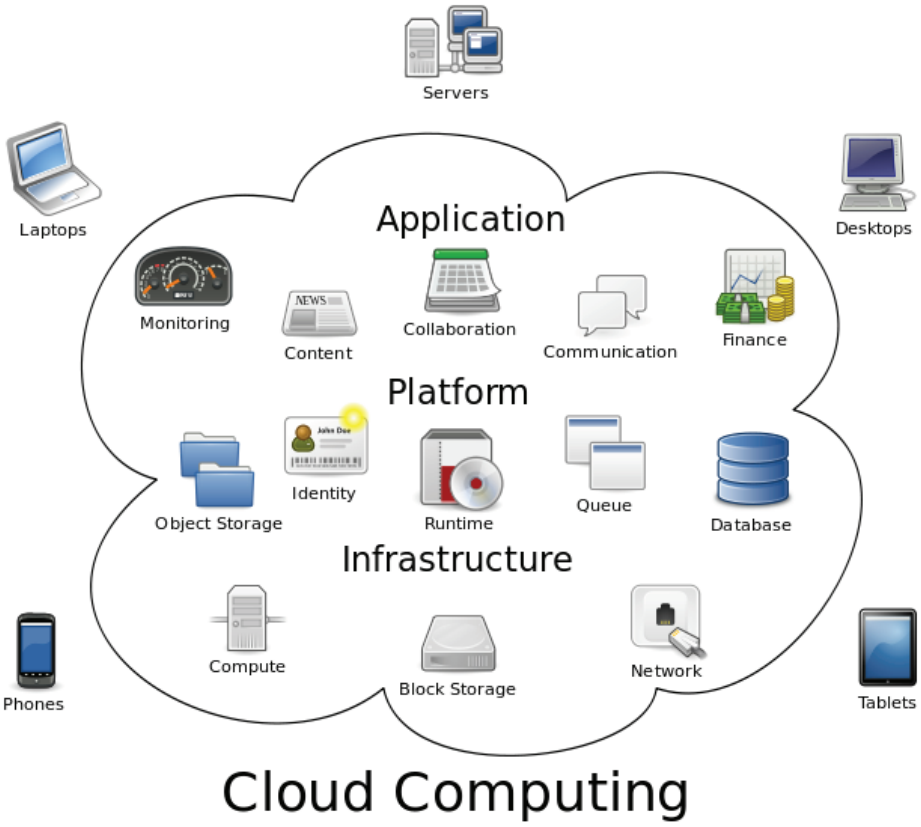


Figure 2.1: Cloud computing logical diagram (excerpt from [6])

In [7], NIST also summarizes the essential five characteristics of cloud computing as follows.

- **On-demand self-service**
  A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.
- **Broad network access**
  Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).
- **Resource pooling**
  The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of

abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.

- **Rapid elasticity**

  Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

- **Measured service**

  Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

Most of these essential characteristics indicate that the cloud computing has more dynamic nature than previous information systems. For example, on-demand self-service nature and elasticity which can change usages and structure of (virtual) systems can make it difficult for system administrators to understand the current situation. The difficulties in cloud system management is discussed in Section 2.3.

NIST also defines three types of cloud service models and cloud deployment models as follows.

Cloud service model

- **IaaS (Infrastructure as a service)**

  The IaaS service provides computing resources such as CPU time, storages and networks. The resources are mainly provided as virtual machines (VM). The cloud users can deploy operating systems and applications on the VMs. While the users do not have control over the cloud infrastructure, they can manage operating systems and applications.

- **PaaS (Platform as a service)**

  The PaaS offering provides environments for application development and runtime, including programming language, its libraries and developing tools. While the cloud users do not have control over the lower layers such as hardware settings or operating systems, they can develop and customize their applications within the range of development and runtime environment.

- **SaaS (Software as a service)**

  The SaaS capability provides applications for its users. The users can access to the applications via some interfaces (e.g. web browser and APIs). While they cannot modify the lower layers such as applications or operating systems, they can customize the applications within a certain level.

Cloud deployment model

- **Private cloud**

  The private cloud is used by a single organization (e.g. company and university). This offering is for users who cannot use public clouds for some reasons such as the security for confidential data. While the organization might own and operate the cloud system by itself, it can also outsource the management tasks to some third parties.

- **Community cloud**

  The community cloud is for exclusive use by a specific community sharing some concerns (e.g. mission and policy). The cloud infrastructure can be owned and managed by one or more of the members of the community. The management tasks can be also outsourced to some third parties.

- **Public cloud**

  The public cloud is for open use by the general public. The public cloud infrastructure for the general use is mainly provisioned in the cloud datacenters owned and managed by cloud providers.

- **Hybrid cloud**

  The hybrid cloud is a combination of different types of cloud infrastructure (private, community, or public). For example, some organizations might use public cloud infrastructure for the front-end of their services for some reasons such as the cost and the elasticity of the public clouds. On the other hand, they can also use private cloud infrastructure to store confidential data. Combining them and providing a consistent service is one of the example of hybrid cloud approaches.

In this dissertation, we mainly discuss relatively small private IaaS cloud architecture which can be used by the limited members (e.g. company employees or university students) and provides computing resources as virtual machines.

2.2 Information system management

The process of information system management consists of various activities. In ITIL (Information Technology Infrastructure Library) V3 [8] published by itSMF (IT Service Management Forum)[24], the system management lifecycle is explained by the following five stages shown in Figure 2.2.

1. **Service Strategy stage**

   The tasks in this stage are to recognize requirements for the system from users or system managers, and to plan strategies for service design, development and implementation to fulfill the requirements.

2. **Service Design stage**

   In this stage, it is needed to materialize the strategies and design the service in detail so that the service can be implemented to the system.

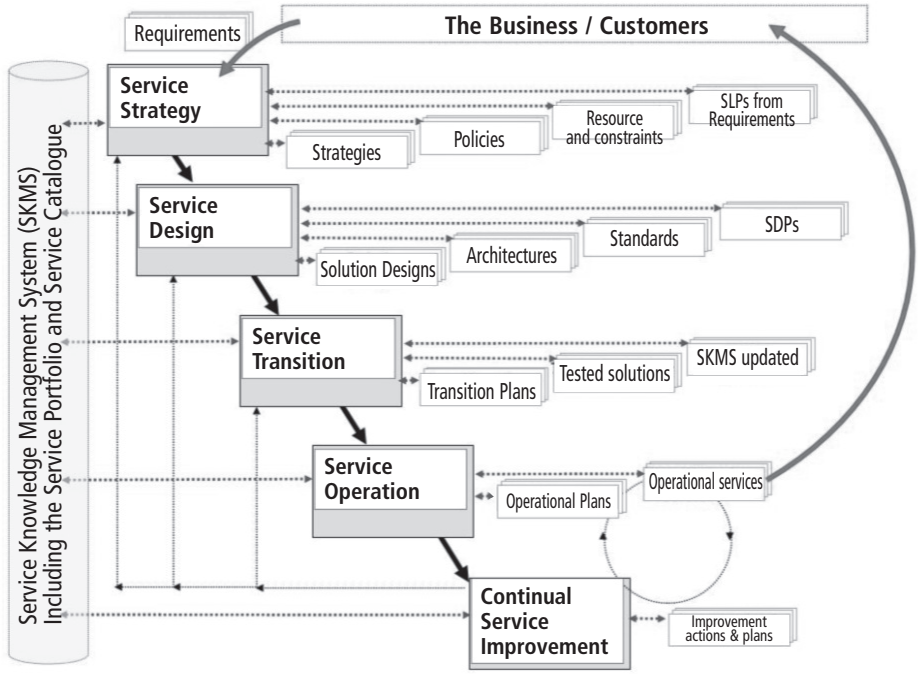


Figure 2.2: Service lifecycle (excerpt from [8])

3. **Service Transition stage**

   Before the activation, the service is verified and evaluated to prevent failures. Then it is implemented and activated.

4. **Service Operation stage**

   Tasks to keep service availability (e.g. problem management) are conducted in this stage.

5. **Continual Service Improvement stage**

   This stage focuses on improving the service performance by assessing some performance indices such as service level, cost and efficiency.

The techniques discussed in the dissertaion is mainly focusing on the service transition stage. In the service transition stage, the changes designed in the service design stage are implemented and materialized. This process mainly consists of the following three activities.

1. **Change Management**

   The activity to ensure that the whole change process is properly executed by conducting tasks such as authorization, evaluation and recording.

2. **Service Asset and Configuration Management**

   The activity to identify and manage the configuration items (CI) in order to keep the integrity in the changes executed to the system.

Table 2.1: Top 10 Obstacles for Growth of Cloud Computing [40]

	Types of obstacle
1	Availability of Service
2	Data Lock-In
3	Data Confidentiality and Auditability
4	Data Transfer Bottlenecks
5	Performance Unpredictability
6	Scalable Storage
7	Bugs in Large Distributed Systems
8	Scaling Quickly
9	Reputation Fate Sharing
10	Software Licensing

3. **Knowledge Management**

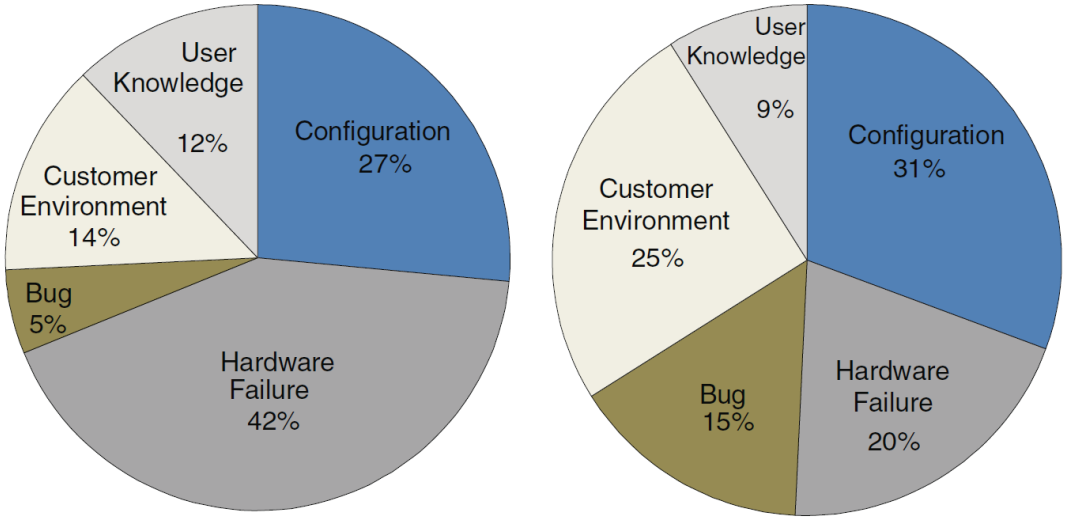
   The activity to manage the service transition to ensure that the required changes are implemented to the system by the right person having the right expertise, at the right time.

The approach proposed in this dissertaion mainly focuses on Service Asset and Configuration Management. In this activity, it is important to keep the integrity while some changes are executed on a system.

2.3 Difficulties in system management

While the best practices described in ITIL and the other frameworks can help system administrators of cloud computing infrastructure, the system administration is quite difficult task. For example, in [40] the top 10 obstacles for growth of cloud computing are listed (Table 2.1). In this list, the most serious concern in using cloud computing is its availability. In reality, the outages of cloud computing services happen every day. While these outages are caused by various types of root-causes, it has been revealed that the most dominant causes of serious problems occured in ICT systems are misconfigurations and improper operations [10] as shown in Figure 2.3. For example, Firstserver Inc. lost business data belonging to 5,700 customers by executing improper operations [41, 42]. Amazon Web Services halted its services because of improper configuration changes to network paths [43].

Although it is quite difficult, cloud vendors need to avoid service faults caused by improper administration to provide stable services to their customers as much as possible, because cloud failures impact on user experiences and the credibility of the cloud vendors. For example, Amazon EC2 (Amazon Elastic Compute Cloud) has to refund 10% of user expenses for penalty if the availability of their services become less than 99.95% [9]. In addition, if a cloud provider causes service fault frequently, it will lost their credibility from users. As a result, its users will cease to use its services and move to other vendors.



(a) Categorization of problem causes on all the cases

(b) Categorization of problem causes on cases with high severity

Figure 2.3: Root cause distribution among the customer problems (excerpt from [10])

However, the structure of data centers has been becoming too complex for administrators to manage by their cognitive capabilities. It is reported that over 66% of data-center staff said their systems were too complex to manage [14]. Therefore, some methods to assist the system administrators' tasks by some technical approaches have been highly required.

2.4 Formal verification

Formal methods [12] are mathematical techniques for software and hardware development. They mainly focus on improving the reliability of system designs or program codes by identifying flaws hidden in them by executing verification. They can be used throughout the process of software/hardware development. Recently, formal methods have been beginning to be applied for the testing and verification of information systems used in our society. For example, in 1998 B-Method[67] was applied to prove safety properties in the control system of automated metro system in Paris. It is also reported that the specification of Felica e-wallet system was described in VDM (Vienna Development Method) formal specification language [11]. The formal specification helped the developers identify the flaws in the specification by executing testing.

The verification approaches in formal methods can be generally categorized in theorem proving or model checking. In the theorem proving, theorems are described as a set of logical formula. Theorem proving tools such as Rodin Platform for Event-B [13] assists users in checking the correctness of refinement using inference rules. It still requires high level expertises to use theorem proving tools. Automatic general theorem proving tools which can be used for various problems have not been developed so far. Based on this situation, we use model checking approach in this dissertation.

2.4.1 Model checking

Model checking [50] is a technique used to check whether a system satisfies given requirements. In this method, first the model representing the behavior of a system is defined by a state transition graph such as Kripke structure represented by the 4-tuple $M = (S, S_0, R, L)$. Here, S is the finite set of states in which the system can exist and $S_0 \subseteq S$ is the set of the initial states of the system. $R \subseteq S \times S$ represents a transition relation between states, and $L : S \to 2^{AP}$ is a labeling function that labels each state with the set AP of atomic propositions whose truth values are true in that state. Next the specification to be satisfied by the system is described as logical formula such as CTL (Computational Tree Logic). Then a model checking tool (e.g. SPIN [52], PRISM [53], and NuSMV [54]) explores the state space to identify the set of states $\{s \in S | M, s \models \psi\}$ in the model M in which the valuation of given logical formula ψ is true. If a sequence of state transition from an initial state to a state in which the valuation of the given formula ψ is false is identified, the model checking tool outputs the sequence as a counterexample for ψ. If not, we can conclude that the specification ψ is satisfied in the model M.

Various types of model checking approaches have been proposed so far. They can be generally categorized in finite-state model checking or bounded model checking. The typical realizations of the former include symbolic model checking using BDD (binary decision diagram) which is used in tools such as NuSMV, and explicit-state on-the-fly model checking used in tools such as Spin. The typical realizations of the latter are to resolve problems into SAT problems as realized in tools such as Alloy. Satisfiability problem (SAT) is a problem to determine whether or not the valuation of a propositional formula can be true by assigning proper truth values to variables in the formula. The SAT problem has been proven to be NP-complete problem. NP-complete problem is defined such that a problem p in NP (Non-deterministic Polynomial time) is also NP-complete if every other problem in NP can be transformed into p in polynomial time. SAT solver is an engine to solve SAT problem and many tools have been developed so far such as miniSAT [28] and SAT4J [27].

As for the specification of requirements, CTL (Computational Tree Logic) and LTL (Linear Temporal Logic) are widely used. In this dissertation, we use CTL formula. CTL[69] is a branching-time logic in which a model of time is a tree-like structure. Therefore, there are different paths in the future and the valuation of a CTL formula is determined by the computational tree from initial states. CTL formula is defined recursively as follows.

- An atomic proposition $p \in AP$ is CTL formula. An atomic proposition has its truth value (either true or false).
- If f_1, f_2 are CTL formulae, $f_1 \wedge f_2$, $f_1 \vee f_2$, $\neg f_1$, $f_1 \to f_2$ are also CTL formulae. The truth value of $\neg f_1$ is the opposite of f_1 (if f_1 is true, then $\neg f_1$ is false, and vice versa). The semantics of the rest formulae is defined by the truth value assginments shown in Table 2.2.
- If f_1, f_2 are CTL formulae, $\mathbf{EX}f_1$, $\mathbf{EF}f_1$, $\mathbf{EG}f_1$, $\mathbf{AX}f_1$, $\mathbf{AF}f_1$, $\mathbf{AG}f_1$, $\mathbf{E}(f_1\mathbf{U}f_2)$, $\mathbf{A}(f_1\mathbf{U}f_2)$ are also CTL formulae.

The semantics of CTL operators can be explained as follows. Here, we denotes $(s, s') \in R$ when the system can change its state from s to s'. We also represent the set of paths from a state s by $Path(s)$.

Table 2.2: Semantics of logical operators

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$
true	true	true	true	true
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

$$Path(s_0) = \{(s_0, ..., s_i, s_{i+1}, ...) | \forall i \in N, (s_i, s_{i+1}) \in R\}$$

Here we suppose that π is an infinite sequence $s_0, ..., s_i, ...$ with $\pi(i) = s_i$. For a state s in a Kripke structure $M = (S, S_0, R, L)$, an atomic proposition $p \in AP$ and CTL formulae f_1 and f_2, the semantics of relation $\models$ is inductively defined as follows.

- $M, s \models p \Longleftrightarrow p \in L(s)$
- $M, s \models (\neg p) \Longleftrightarrow p \notin L(s)$
- $M, s \models f_1 \wedge f_2 \Longleftrightarrow M, s \models f_1$ and $M, s \models f_2$
- $M, s \models f_1 \vee f_2 \Longleftrightarrow M, s \models f_1$ or $M, s \models f_2$
- $M, s \models f_1 \rightarrow f_2 \Longleftrightarrow M, s \not\models f_1$ or $M, s \models f_1 \wedge f_2$
- $M, s \models$EX $f_1 \Longleftrightarrow$ there exists a path $\pi \in Path(s)$ such that $M, \pi(1) \models f_1$
- $M, s \models$AX $f_1 \Longleftrightarrow$ for all paths $\pi \in Path(s)$, $M, \pi(1) \models f_1$
- $M, s \models$EF $f_1 \Longleftrightarrow$ there exists a path $\pi \in Path(s)$ and $i \in N$ such that $M, \pi(i) \models f_1$
- $M, s \models$AF $f_1 \Longleftrightarrow$ for all paths $\pi \in Path(s)$, there exists $i \in N$ such that $M, \pi(i) \models f_1$
- $M, s \models$EG $f_1 \Longleftrightarrow$ there exists a path $\pi \in Path(s)$ such that $M, \pi(i) \models f_1$ for all $i \in N$
- $M, s \models$AG $f_1 \Longleftrightarrow$ for all paths $\pi \in Path(s)$, $M, \pi(i) \models f_1$ for all $i \in N$
- $M, s \models$E f_1 U $f_2 \Longleftrightarrow$ there exists a path $\pi \in Path(s)$ and $i \in N$ such that $M, \pi(s_i) \models f_2$ and $M, \pi(s_j) \models f_1$ for every $j < i$
- $M, s \models$A f_1 U $f_2 \Longleftrightarrow$ for all paths $\pi \in Path(s)$, there exists $i \in N$ such that $M, \pi(s_i) \models f_2$ and $M, \pi(s_j) \models f_1$ for every $j < i$

Since the purpose of this dissertation is to analyze the behavior of an information system, we suppose that the verification (analysis) tools should enable users to define and verify the behavior of a system consisting of many components having interactions between each other. While there are several tools which have functions sufficient to cover these requirements, we decided to use Alloy Analyzer and NuSMV because they are widely-used stable tools.

2.4.2 Alloy Analyzer

Alloy [26] is one of existing verification tools utilizing SAT solvers and based on a first order relational logic. It has been developed at MIT which was inspired by the Z specification language and Tarski's relational calculus. Its language for describing a set of structures is a simple and expressive logic based on the notion of relations. By combining first-order logic formula with the following statements, Alloy provides a simple structural modeling language.

- **Signature (`sig`):** Define a set of objects.
- **Facts (`fact`):** Define constraints that are assumed always to hold.
- **Predicates (`pred`):** Define named constraints.
- **Functions (`func`):** Define named functions that return results.
- **Assertions (`assert`):** Define constraints that are intended to follow from the facts of the model. Alloy Analyzer checks the assertions.

The Alloy Analyzer [26] is a software tool to analyze specifications written in the Alloy specifications. It is intended to provide fully automated analysis for the model specifications in the following way. First, the Alloy Analyzer translates Alloy descriptions into a SAT formula represented by a conjunctive normal formula (CNF). Next, the Alloy Analyzer inputs this CNF into a SAT solver that can determine the satisfiability of the CNF. The Alloy Analyzer can use various SAT solvers such as SAT4J [27] written in Java and miniSat [28]. After analysis by the SAT solver, Alloy determines whether there is a model (an assignment of truth values for variables) that makes the interpretation of the SAT formula true, and it outputs instances of that model if they exist. Since the Alloy Analyzer finds models satisfying a given formula, it is called a "model finder." It can also be used to check properties of the model by generating counterexamples for the assertions. Since the models and counterexamples are displayed graphically, we can easily understand the analysis results. In addition, since Alloy Analyzer has the ability of incremental analysis, it can perform analysis incrementally from small scale with small number of objects to large scale.

2.4.3 NuSMV

NuSMV used in Chapter 4 is a symbolic model checker which has been developed in a joint project between ITC-IRST (Istituto Trentino di Cultura in Trento), Carnegie Mellon University, the University of Genoa and the University of Trento. It is an extension of CMU SMV symbolic model checker which is the first BDD-based model checking tool. NuSMV supports the two types of temporal logic specifications; CTL (Computational Tree Logic) and LTL (Linear Temporal Logic). It can verify the safety and liveness properties described as CTL or LTL temporal logical formula. Same as Alloy, NuSMV has several declarations for defining objects and constraints for them.

- **Modules (`MODULE`):** Define a module having its state transitions.
- **Variables (`VAR`):** Define a set of state variables in a module.

- **Assignments (`ASSIGN`):** Define variable assignments. By assigning values to the variables in the next state (with guard conditions), we can define state transitions of a module.
- **Specifications (`SPEC`):** Define constraints (specifications) which should be verified by NuSMV.

The examples of NuSMV codes can be found in Chapter 4 (e.g. Figure 4.7).

2.5 Target domain

In this section, we discuss the target domain of this thesis in detail. As described before, our target of analysis is in the domain of cloud infrastructure management, especially for private IaaS (Infrastructure as a Service) cloud systems. First, the overview of the structure and fundamental componets in private IaaS along with its management are presented in Section 2.5.1. Then, problems in configuration change management to be handled in our proposed method are discussed in Section 2.5.2.

2.5.1 Target: Management of private IaaS cloud system

As described in various reports, it is expected that the requirements from cloud users will be diverse. This trends will spur demand for private cloud system which can be customized for each cloud user's requirements, rather than using large scale public cloud which provides uniform and homogeneous services to a large number of users. For example, research reports [116, 117] forecast that the market of private cloud will quadruple from 2013 to 2017, resulting in 1.4 trillion Yen. The main differences between private cloud and public cloud can be summarized as follows.

- **Security:** By using private cloud, users can secure their confidential data within their own facilities. In the case of public cloud, users might not be able to locate the whereabout of their data, because the location of datacenters are ususally confidential for security reason. It may cause difficulties in the use of public cloud for some users who need to know the location of data for some reasons such as legal requirements.
- **Scale:** Public clouds provide uniform services to a large number of users by utilizing numerous servers, while private cloud may consist of a small number of hardware components.
- **Management:** In public cloud, it can be expected that a large number of highly expertized administrators manage the infrastructure to provide stable services to their users. As for private cloud, some organizations might have to choose system administrators from the members in the organizations. In this case, it is supposed that the administrators of private cloud are less expertized than the ones of public cloud.

From these characteristics, we can easily imagine that administrators of private cloud will face with the difficulties in the cloud management even if the size of their system is

relatively small, because of the lack of their experience and expertise. Therefore, techniques to assist their task will be in high demand. With the increase in demand for private cloud, it is expected that the importance of such techniques also increase.

Next, in order to define the structure of target system (private IaaS cloud system), identifying the structure and constituents of private cloud system is required. We can identify the components comprising a private cloud from various types of integrated private cloud products. For example, the configuration of IBM PureFlex Systems [118], an integrated infrastructure product, can be seen from [119]. The fundamental components of private cloud can be summarized as follows.

- **Compute Nodes:** Servers with CPUs (e.g. IBM POWER-processor or Intel-processor)
- **Networking:** Switches and cables (e.g. Ethernet or Fiber Channel)
- **Storage Nodes:** Nodes with storage devises (e.g. hard disk drive (HDD) or solid state drive (SSD))
- **Power Unites:** Power supplies and exhaust heat fans

In reality, there are many other functional components required to maintain the private cloud infrastructure. For example, a data center needs chillers (cooling systems) such as CoolLoop [120] to remove heat emitted from cloud infrastructures to out of the data center. Security measures such as personal identification systems to prevent someone from intruding in the data center may also be mandatory requirements. However, these functions are mostly related to the buildings of data center rather than ICT systems. Therefore, we decided to concentrate on the fundamental components listed above, which provide resources (computing time, storage area, network bandwidth and power) consumed by cloud customers.

As for the management of these types of cloud systems, various types of tasks must be performed to keep the systems stable and updated. Especially, configuration changes are requested for cloud administrators every day from the reasons as follows.

- Installation of new functions (e.g. releases of new services)
- Improvement of efficiency (e.g. reducing costs for energy [121] or software license [122] by consolidating virtual machines to a small number of servers)
- Recovery from failure (e.g. replacing a faulty component with new one)

We can say that the first two tasks are schedulable, since administrators can determine the schedule regarding when they are going to apply these changes. On the other hand, the recovery from failure is not schedulable in most cases, because once a failure is observed, administrators have to execute some workaround tasks (e.g. investigating root causes and removing them) to recover services as soon as possible. While both types of tasks are equally important, we decided to concentrate on the analysis of the schedulable changes. The primary reason of this desicion is that faulty schdulable changes can invoke service failures, resulting in the needs for executing workaround (unschedulable tasks). In other words, if we can execute schedulable configuration changes properly and successfully, it means that it can contribute to suppress the occurrences of unnecessary unschedulable configuration changes.

2.5.2 Problems in system configuration changes

Before the execution of schedulable changes, change plans should be designed and evaluated. For example, in ITIL framework [8], documented configuration change procedures are reviewed by Change Advisory Board (CAB) consisting of various stakeholders in ICT departments and business departments so that they can confirm that the planned chnages do not have any undesirable and unexpected side effects impacting on the services. However, even if the reviews are conducted by CAB members, it still has the room for misconfigurations caused by planned changes. As described in the previous section, misconfiguration is one of the most dominant factors in the occurrence of serious failures in ICT systems. The typical two reasons of misconfigurations can be summarized as follows.

1. **Improper procedure planning**

   Since the various types of components comprising a cloud infrastructure are closely interconnected with each other, configuration changes to be conducted for the infrastructure should require expertises in various technical domains. For example, [70] shows that the provisioning process of virtual machines requires configuration changes in operating systems, applications, network (VLAN and firewall) and database. The complexity of the system can make it difficult for administrators to design proper configuration steps. As mentioned in [123], the consultations between experts for various technical domains are mandatory in configuration change planning. If the administrators design improper procesures without taking into account necessary preparation or constraint, the planned process does not work as the administrators intended.

2. **Unintentional changes**

   Even if the configuration changes are properly designed, there still is a room for the occurrence of misconfiguration in its execution phase. While various types of misconfigurations (e.g. command typo) can happen, one of the typical examples is the execution of the changes to wrong target. Because of the complexity of cloud infrastructure, misunderstanding of target can easily happen. For example, in the case of failure occurred in Amazon [43], the configuration change for its network was conducted. The configuration change was supposed to shift traffic to the routers in the primary network which have enough capacity. However, the traffic was accidentally routed to the secondary network with small capacity, resulting in the network congestions. Another typical example of the assignment of wrong target is the confusion of test environment and production environment. We can easily imaginze that the outcome could be disastrous if a process for erasing temporal data in testing environments is applied to production environments serving a large number of customers.

In summary, in order to improve the reliability in configuration changes by preventing undesirable outcomes, we need to do following two things: (1) synthesize proper configuration procedure by taking into account the conditions regarding various types of components comprising the systems and (2) avoid the execution of wrong tasks for wrong targets.

2.5.3 Characteristics of properties to be checked

For the application of formal methods for the analysis to avoid improper procedure planning and unintentional chnages, we have to choose the analysis approach based on the characteristics of the target problem. Here we discuss the characteristics of properties to be checked in our problems and appropriate logic to be applied for the analysis. In general, we can choose one of the following typical logical framework: propositoinal logic, first-order logic, higher order logic and the extension of them with temporal operators.

- **Propositional logic:** In the propositional logic, a logical formula representing the properties to be checked is defined by a concatenation of atomic propositions. This logic is suitable for representing the simple relations between atomic propositions (e.g. "if X is true then Y is also true").
- **First-order predicate logic:** In the first-order predicate logic, by using predicate and quantifiers, we can represent the conditions with "for all" or "existential" quantifiers (e.g. "for all x, if $P(x)$ is true, then $Q(x)$ is also true").
- **Higher-order predicate logic:** While predicates in the first-order logic can have only variables for the parameters of predicates, in the higher-order logic a predicate's parameters can be other predicates. For example, if we have predicates P, Q and a variable x, we can use representations such as $P(Q(x))$.
- **Temporal extension:** As described in Section 2.4.2, the temporal extension of predicate logic can represent properties related to state changes by using temporal operators. Therefore, we can define conditions such as "X is eventually satisfied".

Fisrt, since our analysis target (a cloud system) can have some similar types of components (e.g. servers), it would be easiler to use predicate logic than propositional logic. By using predicate logic, we can define constraints which can be applied for all elements belonging to the same category. As for the temporal extension, we have to determine the needs of it based on the characteristics of properties.

Properties in configuration change procedure synthesis

In order to achieve proper configuration change procedure synthesis, we need to take into account declarative constraints as well as procedural constraints (pre- and post-conditions of operations). The declarative constraints define conditions (e.g. a physical server cannot accommodate virtual machines whose required resources exceed the capacity of the physical server) to be kept during the configuration changes, usually defined declaratively as "anti-pattern" or "not to do list" as shown in [113]. While the configuration change procedure itself has dynamic nature, the declarative constraints (e.g. not to do list) are usually related to whether a (static) state of a system satisfies them or not. Therfore, we conclude that first-order logic is sufficient to express the constraints to be checked.

Properties in unintentional changes

Different from the procedure synthesis, in the analysis of the effect of unintentional changes, we have to expect the occurrence of some undesirable events such as accidental shut off of a server. Therefore, to prevent these undesirable events from causing service

failures, we need to evaluate the system's vulnerability (or resiliency) to these undesirable events. A typical example of resiliency measures is the existence of single point of failure. We can say that the system does not have a single point of failure if any single undesirable event (e.g. server fault or misconfigration) cannot cause service failure. The properties to be checked can be represented by the sentences such as "all functions consisting of a service will be alive eventually, after any single undesirable event". We need temporal operators for describing this kind of constraints including conditions such as "eventually". Therefore, we conclude that temporal logic such as CTL or LTL is appropriate for the analysis. Therefore, we decided to use NuSMV for the verification of the properties described in CTL, since NuSMV is one of the most popular open-source verification tool for CTL properties.

Based on this viewpoint, we present the procedure synthesis methods and operational vulnerability evaluation methods in the following chapters. In Chapter 3, we explain a method of configuration change procedure synthesis using model finder. This method synthesizes a procedure satisfying constraints regarding system management represented by first-order logical formula using Alloy Analyzer. By using this method, we demonstrate the synthesis of configuration procedure for consolidating virtual machines to a server for eneregy saving, by taking into account the constraints from administrators of various types of components such as server, network and storage. In Chapter 4, we explain the operational vulnerability evaluation method using model checking. To achieve this, first we define the operational vulnerability scale representing how a system is susceptible to undesirable events. Next we execute verification to determine which vulnerability level the system is in by evaluating the satisfiability of temporal logic formula by using NuSMV model checker. Then we demonstrate the evaluation of vulnerability of a cloud system for the execution of operations with wrong target assignment for virtual machine live migration and monitoring function in high availability cluster structure.

Chapter 3

Configuration change procedure synthesis

3.1 Difficulties in configuration change planning for a system managed by various administrators

As described in the previous chapter, a cloud system can be managed by multiple domain experts (e.g. server, operating systems, network and database). Incessant configuration changes can be requested to the system from these experts to keep up with business requirements. This situation gives the administrators a major challenge, since configuration changes for components closely interrelated with the other components require a coherent combination of area-specific configuration procedures produced by multiple domain experts. The current practice for integrating these procedures involves the group of experts discussing their operations and constraints, in an attempt to identify violations of constraints from one domain by the operations of another (Figure 3.1). Lack of consultation may lead to other experts ignoring this constraint in their planning, and ultimately to communication failures in the re-configured ICT system. Therefore, when designing system configuration procedures, we not only need to derive a sequence of operations satisfying their pre- and post-conditions but also to comply with these constraints to avoid violating them (Figure 3.2). These kinds of configuration procedure designs by experts having discussions are time-consuming and prone to errors. Actually, over 66% of data-center staff said their systems were too complex to manage [14]. For these reasons, methods of synthesizing appropriate procedures for system configurations are in high demand to achieve reliable changes in configurations without incurring failures caused by faulty human planning.

In this chapter, we propose a method of synthesizing the procedure to change system configurations based on collected data on management knowledge about systems and information on system configurations. First, we define and describe the knowledge on system management in this method as a first-order logic formula using the Alloy language [15]. Next, we derive information on current system configurations from a CMDB

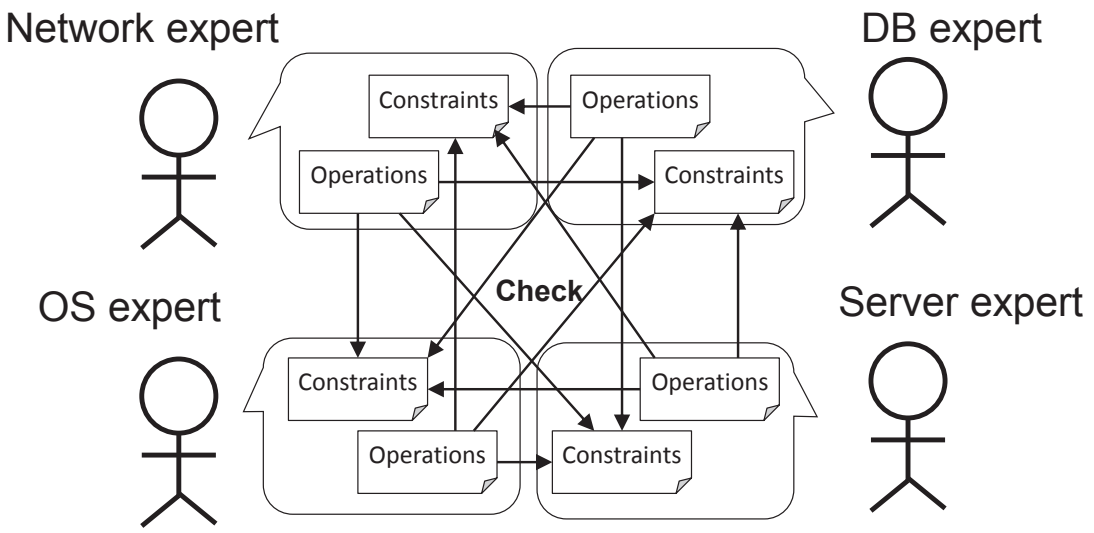


Figure 3.1: Current system configuration procedure designing by discussion between experts

(Configuration Management Database) and translate it into an Alloy description. Then, we combine both knowledge and information and input them into the Alloy Analyzer (a model finder) with goal conditions. The Alloy Analyzer searches for and determines the truth value assignment (valuation) that makes the interpretation of all formulas true. This valuation results in a procedure that leads the system from the initial configuration to that of the target that fulfills the goal conditions. In the procedure synthesis, we identify the intermediate states satisfying some of the set of formulas, and synthesize the procedure between them. By doing this, we can reduce the resources needed in procedure synthesis because Alloy only have to synthesize the procedures for smaller configuration steps instead of synthesizing the whole procedure steps.

The rest of this chapter is organized as follows. First, Section 3.2 presents our method along with how knowledge on system management is represented. Next, we explain how it works through a case study in Section 3.3. After Section 3.4 explains our evaluation, we give some discussions in Section 3.5. Finally, Section 3.6 concludes the chapter and outlines future challenges.

3.2 Procedure synthesis for system configurations

This section explains our method of synthesizing the procedure for system configurations in detail. First, we explain the architecture for our method. Next, we use a case study to explain how information on the system structure and knowledge on system management are represented in it. Then we explain the algorithm for synthesizing configuration procedures using Alloy Analyzer.

There are three main reasons for using Alloy and the Alloy Analyzer in our method of synthesizing the configuration procedure, while other model checkers (e.g. SPIN or NuSMV) can also be used.

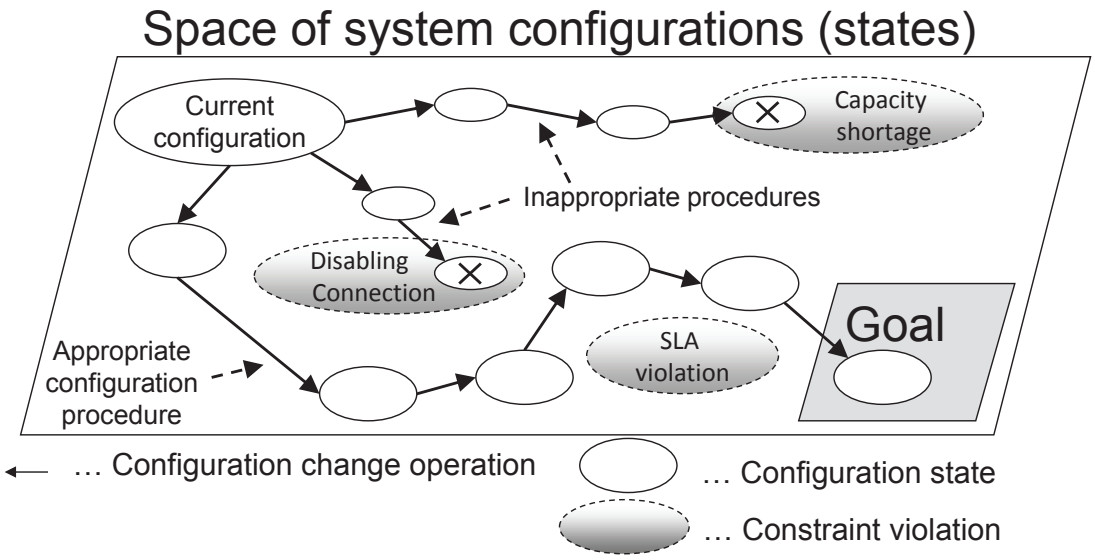


Figure 3.2: Synthesis of procedure for system configurations avoiding violations of declarative constraints

- The Alloy Analyzer enables us to easily find an instance that can satisfy all conditions by just describing them in Alloy language. With the capabilities of SAT solvers, the Alloy Analyzer effectively executes exhaustive exploration of the state space.
- We can define complex conditions in Alloy because of its flexible descriptive capabilities based on first-order logic. It is also good at representing binary relations between variables and transitive closures of relations. This capability is suitable for modeling today's ICT systems that are constructed by connecting various components.
- Changes to system configurations can be defined by the transition relations between the system's states. This enables us to represent system-management operations to dynamically change system configurations.

3.2.1 Architecture

The high-level architecture in our method is depicted in Figure 3.3. Synthesizing a procedure for system configurations with this method involves four elements:

(1) Configuration information from all system components is stored in a Configuration Management Database (CMDB) [23], a storage solution for managing relationships between system components that was proposed in the Information Technology Infrastructure Library (ITIL) [24]. We implemented the CMDB on the AXIS2 [34] server and used Resource Control eXtensible Markup Language (RCXML) [25] for the data format. RCXML is a customized XML format used to integrate system and management information.

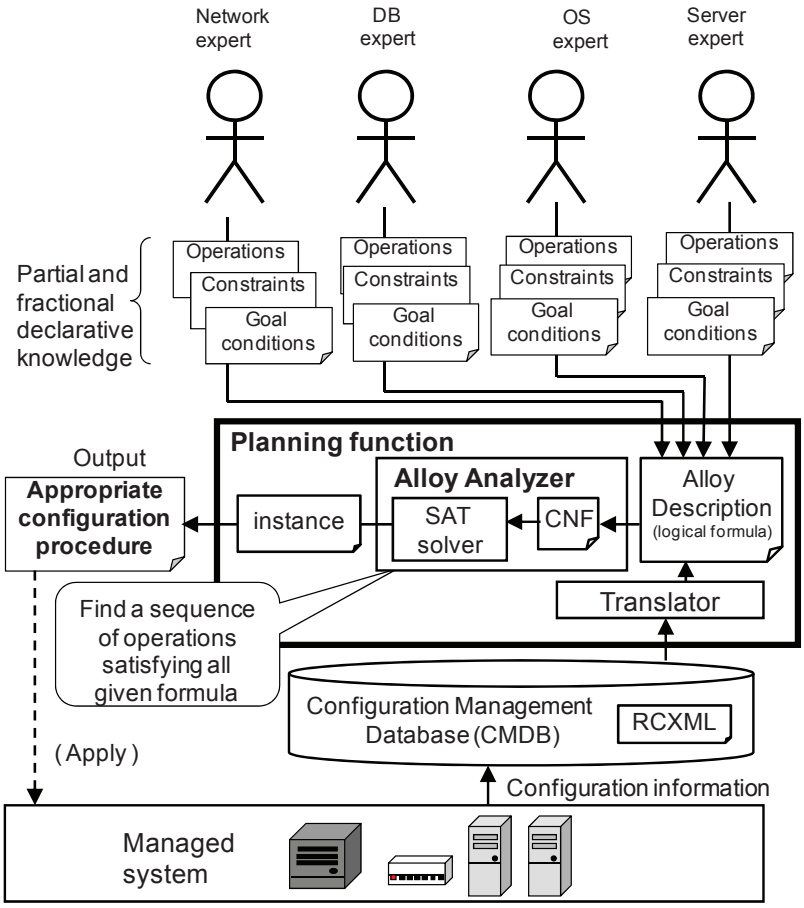


Figure 3.3: Architecture for method of synthesizing procedures

(2) The configuration information (relations between components) stored in CMDB is translated into Alloy descriptions, i.e., first-order logic formulas through a translator. We implemented the translator in Java.

(3) Each expert responsible for managing the components depending on their expertise defines their knowledge on system management, such as constraints and operations with pre- and post-conditions, along with goal conditions in some management tasks, in the Alloy descriptions.

(4) From the information on system configurations and the knowledge on system management, the Alloy Analyzer detects a model (a situation) in which all these formulas written in Alloy are true. We can regard the model generated by the Alloy Analyzer as a synthesized procedure for system configurations that is a sequence of operations leading the system from the current configuration to a state satisfying defined goal conditions without violating any given constraints.

3.2.2 Management knowledge representation

Here, we use a case study to explain how information on the system structure and three types of essential knowledge on system management (executable operations, declarative

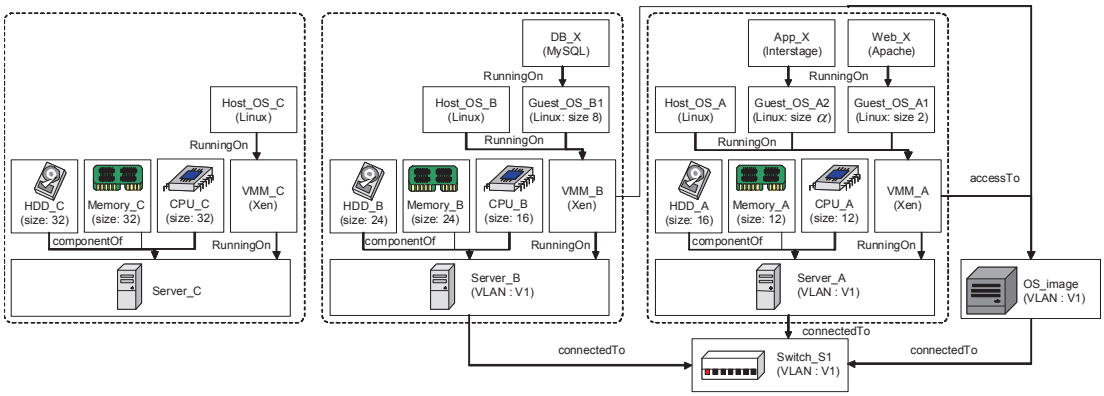


Figure 3.4: Initial system configuration in case study

constraints, and goal conditions) are represented in Alloy. In this case study, we have assumed we are going to derive an appropriate procedure for configuration to consolidate virtual machines onto a server by migration for the managed system in Figure 3.4. As discussed in [70] and [90], various tasks such as network and storage connections, VLAN associations are required for conducting proper provisioning or migration of virtual machines. The case study scenario was designed as one of the examples in which administrators of relatively small cloud systems (e.g. private cloud) need to synthesize a proper configuration procedure which requires domain knowledge regarding various factors comprising the system such as virtualization, network and storage.

System structure

The system we considered in our case study (shown in Figure 3.4) consists of three physical servers (Server_A, Server_B and Server_C) and an OS-image storage device (OS_image). Any subset of these components can be organized into a VLAN by means of a switch (Switch_S1). Each server is running Xen [31] virtual machine monitors (VMMs). On each VMM (VMM_A, VMM_B, and VMM_C), Linux operation systems are running as a host OS or guest OSs. On VMM_A's guest OS (Guest_OS_A1), an Apache Web server (Web_X) is running, while an Interstage application server [32] (APP_X) is running on Guest_OS_A2. Likewise, a MySQL database server (DB_X) is running on Guest_OS_B1 (a guest OS on Server B). We have assumed services are provided to users by a three-tiered system consisting of these three pieces of software (Web_X, APP_X, and DB_X). In addition, Server_C is standing by in case there is any shortage of system capacity. First, while Server_A, Server_B, and OS_image are connected to Switch_S1 where they belong to the same VLAN segment V1, Server_C is not connected to the network.

Each server consists of three types of hardware components: a CPU, memory, and hard disk. These components have a parameter called "size" representing the capacity of resources that can be used to accommodate virtual OSs. We normalize the parameters to a value ranging from 0 to 32 units according to their performance and capacity. For example, in the price list of Amazon EC2 [33] services, the smallest set of resources provided by the service consists of one processor (Intel Xeon or AMD Opteron) with a

```
<Components>
 <Hardwares>
   <Servers>
     <Server id="Server_A">
      <Configuration>
       <CPU id="CPU_A"       size="12" />
       <Memory id="Memory_A" size="12" />
       <HardDisk id="HDD_A"  size="16" />
       <Link id="svrA_link1" src="CPU_A"    dest="Server_A" type="componentOf" />
       <Link id="svrA_link2" src="Memory_A" dest="Server_A" type="componentOf" />
       <Link id="svrA_link3" src="HDD_A"    dest="Server_A" type="componentOf" />
      </Configuration>
    </Server>
...
```

Figure 3.5: Definition of system configuration

clock frequency of 1.0 to 1.2 GHz and a memory of 1.7 GB. We can define the value of the size parameter by assuming that the size of a component is one unit when the component has sufficient capacity to accommodate a certain OS requiring the smallest set of resources. We also assumed that each OS had a size parameter and that a server could not accommodate OSs if the total size of these OSs exceeds the size of any of the server's hardware components. For example, size α (a parameter in the case study) of Guest_OS_A2 should not be more than 10 units, because the size of Guest_OS_A1 is two units and the size of Server_A's CPU (CPU_A) and memory (Memory_A) are both 12.

The configuration information in our method is stored in CMDB in RCXML format, as shown in Figure 3.5. This figure shows the physical configuration of Server_A consisting of three physical components (CPU, memory, and hard disk) by connecting the elements representing these components with the Server_A element via "componentOf" links. Figure 3.6 shows definitions of various relations in the case study. "Link" elements with attributes such as "connectedTo" or "runningOn" represent relations between components. The fact that some components belong to VLAN segment V1 can also be defined by "VLANs" type connections between these components and the element representing VLAN V1.

Our method accesses the configuration information stored in CMDB by using Xpath queries, and translates it into relation definitions in Alloy. Figure 3.7 shows part of the configuration information of the system in the case study translated from RCXML to Alloy by our translator. The relations in this figure, which can be used to characterize the configuration status, are defined in `Sig State {}` descriptions. A `Sig` declaration is used to introduce a set of atoms along with a set of relationships between atoms. For example, there is a set of unidirectional relations defined (defined as the `componentOf` relation) from hardware components (CPU, memory, and hard disk) to servers in a system-configuration state. Along with these relation definitions, the system's initial-configuration status derived from CMDB is translated into `fact {}` declarations. The `fact` declarations define facts that are assumed to hold in a state. For example, the equality starting with `first.componentOf` defines the fact that the `componentOf` relationships (CPU_A $\rightarrow$ Server_A) and (Memory_A $\rightarrow$ Server_A) are held in the initial (first) configuration of the system. The sizes of components are also translated into relationships between these components and integer values as shown in the `first.size` equation.

```
<Links>
 <Link id="ctl1" src="Server_A" dest="Switch_S1" type="connectedTo" />
 <Link id="ctl2" src="Server_B" dest="Switch_S1" type="connectedTo" />
 <Link id="ctl3" src="OS_image" dest="Switch_S1" type="connectedTo" />
...
 <Link id="osl1" src="Host_OS_A"   dest="VMM_A" type="runningOn" />
 <Link id="osl2" src="Guest_OS_A1" dest="VMM_A" type="runningOn" />
...
 <Link id="acl1" src="VMM_A" dest="OS_image" type="accessTo" />
 <Link id="acl2" src="VMM_B" dest="OS_image" type="accessTo" />
...
 <Link id="btl1" src="Server_A"  dest="V1" type="VLANs"/>
 <Link id="btl2" src="Server_B"  dest="V1" type="VLANs"/>
 <Link id="btl3" src="Switch_S1" dest="V1" type="VLANs"/>
 <Link id="btl4" src="OS_image"  dest="V1" type="VLANs"/>
</Links>
```

Figure 3.6: Definitions of relations between components (part)

Executable operations

We assumed that four operations could be executed in the system (Figure 3.8).

Connection operation: We can establish a physical network connection between any server and network device (e.g., a switch) with an Ethernet cable.

Access config operation: We can modify some configuration files of servers to allow one server or piece of software to access another one.

VLAN config operation: We can change the VLAN configurations of servers or network devices to make them belong to some VLAN segments.

Migration operation: We can move a virtual OS from one VMM to another by using the migration function of virtual machines under the condition that both physical servers accommodating these VMM can access the same OS_image storage.

We define knowledge about these operations by relationships that should hold before or after operations so that Alloy Analyzer can determine changes in configurations triggered by these operations. Figure 3.9 has the definitions of knowledge of the four operations by using a predicate declaration in Alloy. In these operation definitions, system configuration states `s` before an operation and state `s'` after the operation is executed are used to define the relationships held in these states. For example, the definition of the connection operation for components `src` and `dst` represented by `connect [s,s':State, src,dst: Objects]` shows that the following three facts hold.

(a) No `connectedTo` relation from `src` to `dst` before connect operation is executed

(b) No `connectedTo` relation from `dst` to `src` before connect operation is executed

(c) The set of `connectedTo` relations after the connect operation is the union of the set of ones before the operation and `(src -> dst)`

```
/ *** System's state definition ***/
sig State {
size :       (CPU ->  Int ) +(Memory ->  Int ) +(HardDisk ->  Int ) +(OS ->  Int ) ,
connectedTo: (Server -> NetworkDevice) + (NetworkDevice -> Server) +
             (NetworkDevice -> NetworkDevice),
componentOf: (CPU + Memory + HardDisk) -> Server ,
accessTo:    (Program + OS + Server) -> (Program + OS + Server) ,
runningOn:   (Program + OS ) -> (Program + OS + Server) ,
...}

/ *** Initial state configuration ***/
fact {
  first.size = (CPU_A -> Int[12]) + (CPU_B -> Int[16]) + (CPU_C -> Int[32]) +
  (Memory_A -> Int[12]) + (Memory_B -> Int[24])+( Memory_C -> Int[32]) + ...

  first.componentOf = (CPU_A -> Server_A) + (Memory_A -> Server_A) + ...
  first.accessTo    = (VMM_A -> OS_image) + (VMM_B -> OS_image)
  first.runningOn   = (Host_OS_A -> VMM_A) + (Guest_OS_A1 -> VMM_A) + ...
...}
```

Figure 3.7: System configrations translated from RCXML to Alloy (part)

The rest of the operations are defined in the same way. In the definition of the migration operation for virtual machines, both `src` and `dst` must be Xen and the OS_image must be accessible from them as preconditions of the operation. Note that we not only need to define the relationships that have changed after operations but also explicitly specify the frame conditions [36] stating that the relationships not mentioned in these operations have not changed. Without the frame conditions, model finders might consider that relationships that have not been mentioned can implicitly be changed. This can result in irrelevant procedures being output containing impossible changes to configurations. In our method, we accomplish the frame-condition description with flag description `s'.changes=*` added to the last part of each operation definition; the relations between the flags and the system status changes are defined in Figure 3.10. For example, in the definition of the connection operation in Figure 3.9, the description, `s'.changes = connectedTo_c`, clarifies that the operation only changes `connectedTo` relations by including the flag, `connectedTo_c`, in the set of flags `s'.changes`. At the same time, the frame condition, "`(s.connectedTo = s'.connectedTo || connectedTo_c in s'.changes)`", described in Figure 3.10 defines that unless this flag is included in the set `s'.changes`, the `connectedTo` relation remains the same in state `s'` after the operation.

The system's state transitions (possible configuration changes) triggered by these operations can be defined by using both the operation knowledge and the frame conditions as shown in Figure 3.11. This description means that if there are some components `x`, `y`, and `z` satisfying one of these predicates representing the operations and the frame conditions, the system can change its status (configuration) from `s` to `s'`.

Constraints (requirements)

We also assumed that the following accessibility and VLAN constraints were defined by network management experts, and the capacity constraint by virtual machine management experts.

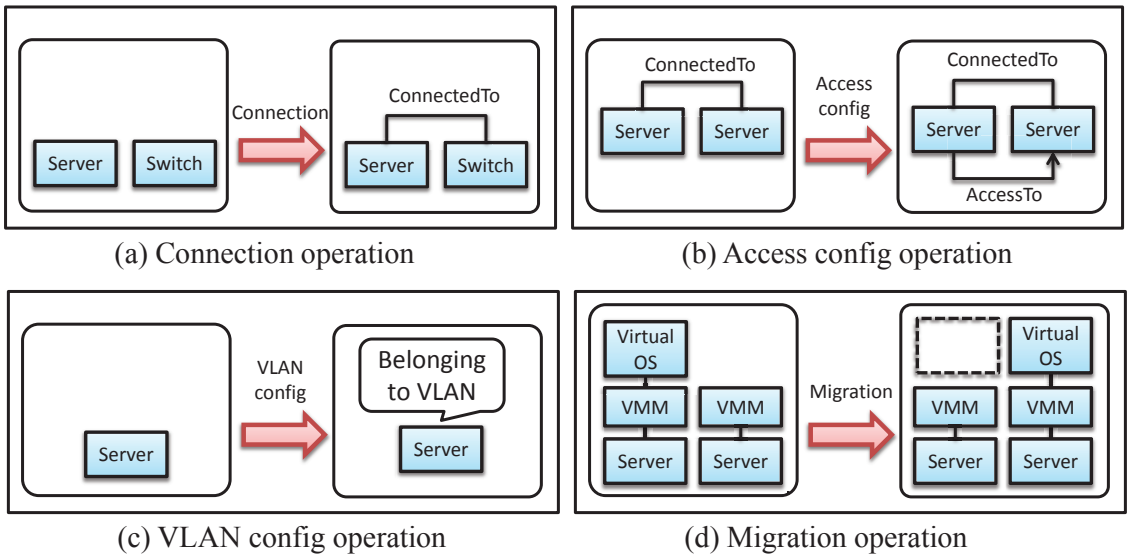


Figure 3.8: Executable operations

Accessibility constraint: In order for two network components to be able to access each other, these components should be connected via some network links.

VLAN constraint: In order for two network components to be able to access each other, both of these components should belong to the same VLAN segment, or neither of them should belong to any VLAN segment.

Capacity constraint: The total size of OSs on a server should not be more than the size of any of the server's hardware components.

We can define these constraints to be retained in systems by describing them in first-order logic formulas in Alloy's `fact {}` declarations shown in Figure 3.12. For example, the accessibility constraint is defined by the fact declaration stating that if there is an `accessTo` relation between some components `x` and `y` in state `s`, then `y` should be able to be reached from `x` through a set of `runningOn` and `connectedTo` relations and their inverse relations in the state. In the Alloy description, we can represent the transitive and reflexive closure of a relation and the inverse of a relation by using an asterisk (*) for the former and a tilde (~) for the latter. The concatenation of different types of relations can be represented by a dot (.). The VLAN and capacity constraints are also defined in the same way in Figure 3.12.

Goal conditions (Request for change)

Here, we have assumed that there is a request to consolidate all pieces of software (Apache, Interstage, and MySQL) comprising the service into the same physical server to conserve energy by shutting down servers that are unused. Note that which server we should use to accommodate these pieces of software on by migration is determined depending on the current system configuration. For example, if the capacity, α, of Guest_OS_A2 is 2, we can achieve this goal just by migrating Guest_OS_B1 from VMM_B to VMM_A, since the sum of the OSs' sizes is 12 (2+2+8) and all hardware components on Server_A have

```
/*** Operation Definitions ***/

/* Connection operation: Connect src and dst */
pred connect[s,s':State, src,dst: Objects]{
  (not ((src->dst) in s.connectedTo)) &&
  (not ((dst->src) in s.connectedTo)) &&
   s'.connectedTo = s.connectedTo + (src->dst)&&
   s'.changes = connectedTo_c }

/* Access config operation: make src access to dst */
pred addaccessTo [s,s':State, src,dst: Objects]{
   not ((src->dst) in s.accessTo)) &&
  s'.accessTo = s.accessTo + (src -> dst) &&
  s'.changes = accessTo_c }

/* VLAN config operation: make src join dst */
pred joinVlan [s,s':State, src,dst: Objects]{
  s'.VLANs = s.VLANs + (src -> dst) &&
  s'.changes = VLANs_c }

/* Migration operation: migrate vm from src to dst */
pred migrate[s, s':State, vm,src,dst: Objects]{
   (Xen in (src.(s.name) & dst.(s.name)) ) &&
   (OS_image in (src.(s.accessTo)
           & dst.(s.accessTo)) ) &&
   ((vm->src) in s.runningOn) &&
    s'.runningOn = s.runningOn ++ (vm->dst) &&
    s'.changes = runningOn_c
}
```

Figure 3.9: Definitions of operations in Alloy

sufficient capacities to accommodate them. If α is more than 2, on the other hand, we cannot consolidate them into Server_A due to the capacity constraint. In addition, if we consolidate them into Server_B or Server_C, the configuration procedures are completely different from the one used for consolidation into Server_A. Therefore, we need to derive both a configuration satisfying the goal conditions and an appropriate sequence of operations that can change the system configuration from its initial configuration to the one required.

We can define the goal conditions to be satisfied in the state after an appropriate sequence of operations is executed by using `fact {}` declarations in the same way as declarative constraints. Figure 3.13 shows the goal conditions for the case study (con-

```
/*** Frame conditions ***/

pred frame_condition [s,s':State]{
(s.connectedTo = s'.connectedTo || connectedTo_c in s'.changes) &&
(s.componentOf = s'.componentOf || componentOf_c in s'.changes) &&
(s.accessTo = s'.accessTo       || accessTo_c in s'.changes)    &&
...}
```

Figure 3.10: Definitions of operations in Alloy

```
/*** State Transition Definitions ***/

fact StateTransition { all s: State, s': ord/next[s] |
 ( (some disj x,y: (s.Server + s.NetworkDevice) | connect[s,s',x,y])      ||
   (some disj x,y: (s.Program + s.OS + s.Server) | addaccessTo[s,s',x,y]) ||
   (some x: (s.Server + s.NetworkDevice) | some y: s.VLAN| joinVlan[s,s',x,y])    ||
   (some x: s.OS | some disj y,z: s.Program | migrate[s,s',x,y,z])     )
       && frame_condition [s,s'] }
```

Figure 3.11: Definitions of state transitions with operation knowledge and frame conditions

```
/*** Declarative constraints ***/

/* Constraint 1
   If x has access to y, x and y should be connected */

fact {all s: State | all disj x,y: (s.Program + s.OS + s.Server) |
  (x->y) in s.accessTo => y in x.*(s.runningOn).
         *(s.connectedTo + ~(s.connectedTo)). *(~(s.runningOn))}

/* Constraint 2
   If x has access to y, x's and y's server should belong to the same VLAN */

fact {all s: State | all disj x,y: (s.Program +s.OS + s.Server) |
   (x->y) in s.accessTo => (no (x+y).*(s.runningOn).(s.VLANs)) ||
        y in x.*(s.runningOn).(s.VLANs).~(s.VLANs).*(~(s.runningOn))}

/* Constraint 3:
   Total OSs size should be less than or equal to components' capacity */

fact {all s: State | all x: s.Server | all c: x.(~(s.componentOf)) |
    c.(s.size) >= (sum y: x.^(~(s.runningOn)) | (y & s.OS).(s.size) )}
```

Figure 3.12: Declarative constraints

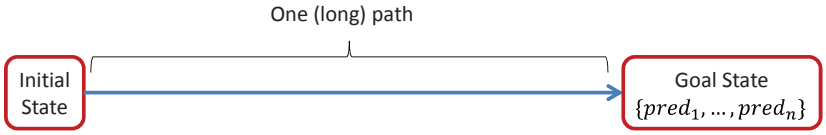
solidation of Web_X, App_X, and DB_X) defined by the `fact` declaration that there is state `s` where there is server `x` on which all of Web_X, App_X, and DB_X are running. In this figure, the hats (^) represent the transitive closure of relations without any reflexive relations.

3.2.3 Procedure synthesis algorithm

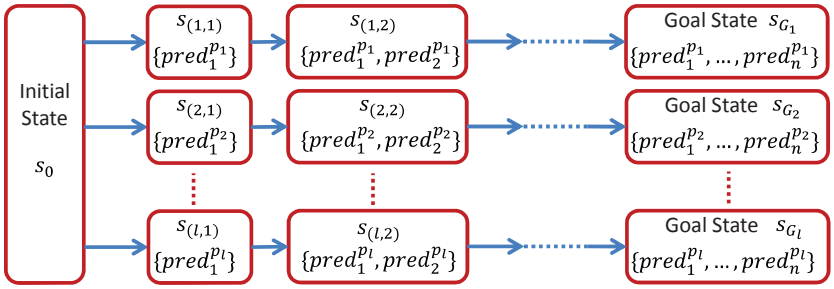
In order to synthesize the configuration change steps from initial state to the state satisfying the goal condition, we input the required information (initial system configuration, constraints, executable operations, and goal conditions) to Alloy Analyzer model finder. Then Alloy Analyzer can derive the state changes from initial state to the goal state. However, in the state search, we have to give a finite upper limit to the number of steps to be searched in order to avoid state space explosion. The simplest approach in controlling the number of steps is to increment the upper limit one by one until we find a state satisfying the goal condition (Figure 3.14(a)). However, long configuration steps can lead the increase of the number of SAT clauses constructed from the system model, resulting

```
/*** Goal condition ***/
fact { some s: State | one x: s.Server |
   x in Web_X.^(s.runningOn) && x in App_X.^(s.runningOn) && x in DB_X.^(s.runningOn)
}
```

Figure 3.13: Goal condition



(a) Straightforward search path



(b) Multiple short search path

Figure 3.14: State search with intermediate states

in the large memory size used for the configuration procedure synthesis. Therefore, we need to keep the number of configuration change steps as small as possible in the state space search by Alloy Analyzer.

In order to solve this problem, we divide the search path from the initial state to goal state by "intermediate goal states" satisfying a part of goal conditions, as shown in Figure 3.14(b). Then we synthesize the procedure from the initial state to the goal state through these intermediate states. Here we only have to synthesize some short paths of whole required steps and concatenate those parts, instead of making Alloy Analyzer synthesize the whole procedure. By this approach, our approach decomposing the state space search into some smaller parts can synthesize the procedure with smaller memory.

The algorithm of configuration procedure synthesis utilizing the intermediate goal states consists of two parts: 1) identifying the concrete goal states and intermediate goal states and 2) searching for partial procedures from source states (initial or intermediate states) to destination states (intermediate or goal states) by preferring the shortest paths between them. The explanation for each step is as follows.

(1) Identify the goal and intermediate states

Here we first derive "concrete goal conditions" by assigning possible values for variables $X = (x_1, x_2, ..., x_m)$ in the set of predicates $Pred = (pred_1, pred_2, ..., pred_n)$ which are

to be satisfied to achieve goal condition. We represent the possible value assignments for x_i by $A(x_i) = \{C_{i_1}, C_{i_2}, .., C_{i_{l_i}}\}$ and all combination of the value assignments by $P = \{p_1, p_2, ...p_l\} = A(x_1) \times A(x_2) \times ... \times A(x_m)$. We represent the predicate $pred_i$ with an assignment p_j by ${pred_i}^{p_j}$. We define the set of concrete goal conditions $CGC = \{cgc_1, cgc_2, ..., cgc_l\}$ where $cgc_i = \{{pred_1}^{p_i}, {pred_2}^{p_i}, ..., {pred_n}^{p_i}\}$.

Next, we define the state transition model $M = (S, T, L)$ to represent the initial configuration state, the concrete goal states satisfying the concrete goal conditions and the transition relation between them. Here, S is a finite set of states representing a configuration. $T \subseteq S \times S$ represents a transition relation between states which can be invoked by a proper configuration change operation. $L : S \to 2^{Pred} \times P$ is a labeling function which labels each state with the set of predicates with an assignment satisfied in that state. We also have an initial state $s_0 \in S$ representing the initial condition and the set of concrete goal states $S_G = \{s_{G_1}, s_{G_2}, ..., s_{G_L}\} \subseteq S$ satisfying the concrete goal condition $(L(s_{G_i}) = cgc_i)$. Then we define intermediate states $s_{(i,j)} \in S$ with $L(s_{(i,j)}) = \{{pred_k}^{p_i} | k \leq j\}$ representing the states where j out of n predicates in cgc_i under the assumption that the order of predicates to be satisfied does not affect the reachability to the goal state. If we obtain two states $s_{(i_1,j_1)}$ and $s_{(i_2,j_2)}$ having the same set of predicates with the same assignment $(L(s_{(i_1,j_1)}) = L(s_{(i_2,j_2)}))$, we eliminate one of them from S because these are redundant. As a result of this step, the path to be investigated described in Figure 3.14(a) can be modified to the paths having a set of state $\{s_0, s_{(i,1)}, s_{(i,2)}, ..., s_{(i,n-1)}, s_{G_i}\}$ on it, as shown in Figure 3.14(b).

(2) Search for the procedure between intermediate states

After identifying the state transition model, we synthesize the operation procedure by determining the number of operation steps required to transit between these states by Alloy Analyzer. The search algorithm based on the breadth-first search algorithm taking into account the number of steps between intermediate states is summarized in Figure 3.15. In this algorithm, we first set the initial lower bound $len(s, s')$ of estimated number of steps between s and s' to zero, because it is possible that all of predicates $L(s')$ to be satisfied in s' have been already satisfied in the previous state s. Next, we choose a transition for which we check the reachability. In order to make state space to be searched small, we choose a transition (s_{i^*}, s_{j^*}) with the smallest estimated lower bound to reach one of the concrete goal states. Then we check if the destination s_{j^*} is reachable from the source s_{i^*} within $len(s_{i^*}, s_{j^*})$ steps by Alloy Analyzer. If it is reachable, we add new paths $(s_{j^*}, .)$ to the set of paths which are the candidate of next search target, and choose the next path to be searched. Otherwise, we increment the estimated lower bound $len(s_{i^*}, s_{j^*})$ by one and choose the next transition. We repeat this search until reaching one of the concrete goal states. By concatenating the state transitions on the paths derived by Alloy Analyzer, we can obtain the configuration procedure to achieve the goal conditions. We also prepare a parameter n representing the upper limit of the path length. If the estimated lower bound $len(s, s')$ becomes larger than the limit n for any s and s', we conclude that the procedure is too long to be synthesized by our approach.

Remark

```
INPUT : M = (S, T, L),n
OUTPUT : A path from initial to a goal state
Open := {(s_0, s_i)|(s_0, s_i) ∈ T};
d(s) := 0, for all s ∈ S;
len(s_i, s_j) := 0 for all (s_i, s_j) ∈ T;

While true
{
  find (s_i*, s_j*) such that
    len(s_i*, s_j*) = min{len(s_i, s_j)|(s_i, s_j) ∈ Open};
  Alloy checks if s_j* is reachable
    from s_i* within len(s_i*, s_j*) steps;
  If reachable
  {
    Record the operation steps from s_i* to s_j*;
    d(s_j*) := d(s_j*) + len(s_i*, s_j*);
    Remove all (;s_j*) from Open;
    Open := Open ∪ {(s_j*), s_k)|(s_j*, s_k) ∈ T};
    len(s_j*, s_k) := 0 for all (s_j*, s_k) ∈ T;
    If s_j* ∈ S_G { break; } /* Solution found */
  }
    else { len(s_i*, s_j*) := len(s_i*, s_j*) + 1;}
  If len(s_i*, s_j*) > n { break; } /* Solution not found in n steps */
}
```

Figure 3.15: Procedure synthesis algorithm

Here we discuss soundness and completeness of our approach using intermediate goals by comparison with the straightforward approach.

1. Soundness

   Here we suppose the soundness condition is that our approach does not find procedures if they cannot be found by the straightforward approach. In this case, there is no variable assginment satisfying the goal condition in the straightforward approach. In our approach, we derive concrete goal conditions by giving all possible variable assginments to the goal condition. Therefore, if the goal condition in the straightforward approach cannot be satisfied by any variable assignment, any concrete goal conditions also cannot be satisfied. As a result, our approach cannot find any procedure which cannot be found by the straightforward approach, which means our approach satisfies the soundness condition.

2. Completeness

   Here we suppose the completeness condition is that our approach can find a procedure if the straightforward approach can find one. In this case, there is at least one variable assignment which can satisfy the goal condition in the straightforward approach. It means that at least one concrete goal condition can be satisfied in our approach. We can also say that the reachability to the concrete goal states is not affected by the order of the intermediate goal states even if there are interdependencies between predicates to be satisfied in goal states. For example, suppose that in a concrete goal state s_g two predicates ${Pred_A}^{p_1}$ and ${Pred_B}^{p_1}$ with the assignment p_1 are true. We also suppose that we define an intermediate goal states s_1 as a state

satisfying $Pred_A{}^{p_1}$. If $Pred_A{}^{p_1}$ is a prerequisite to satisfy $Pred_B{}^{p_1}$, we can derive a procedure from the initial state to the concrete goal state s_g via the intermediate goal state s_1. On the other hand, if $Pred_B{}^{p_1}$ is a prerequisite for $Pred_A{}^{p_1}$, first our approach searches for a procedure from the initial state to s_1. If it finds one, we can also see that $Pred_B{}^{p_1}$ (prerequisite for $Pred_A{}^{p_1}$) is satisfyied in s_1. In this case, the procedure from the initial state to the goal and the one to the intermediate goal are the same ($s_1 = s_g$). Therefore, we can say that our approach can find a procedure if the straightforward approach can find one. From this discussion, we can conclude that our approach is complete, while the operation steps in the procedures found by both approaches can be different.

While we can say that our approach is sound and complete with the straightforward approach from the above discussion, note that the algorithm does not guarantee that it identifies the shortest path to the goal state. In order to find the procedure with smaller steps, it is preferable that the administrators sort the order of the predicates in $Pred$ so that the predicates with low numbers are pre-conditions for the predicates with high numbers based on their operation knowledge.

3.3 Example synthesis of procedure for configurations

Here, we present an example of synthesizing the procedure for configurations with our method using the case study described in the previous section. The goal of synthesizing the procedure in this scenario is to obtain a sequence of operations that can change the system configuration from the initial arrangement described in Figure 3.4 to one where the three pieces of software Web_X, App_X, and DB_X are consolidated on the same server by migration. To demonstrate that our method can derive appropriate procedures for different initial configurations, we executed our synthesis of the procedure in three cases in which we set size α of Guest_OS_A2 on which App_X was running to 2, 4, and 8. We used the system configuration information, the executable operations, the declarative constraints, and the goal conditions described in the previous section.

We implemented our synthesis program in Java and executed the synthesis of the procedure on a PC with an Intel Xeon 3-GHz CPU, a 4-GB memory, and a 32-bit Windows 7 OS. The program calls the Alloy Analyzer's API. Then, the Alloy Analyzer scans given Alloy descriptions and searches for a sequence of states that can satisfy all given constraints and the sequence of operations triggering the state transitions. We used Sat4J as a SAT solver called from the Alloy Analyzer.

As a result of our synthesis of the procedure for all cases, we obtained the change sequences for system configurations shown in Figure 3.16. Figure 3.17 shows an example of an output screenshot representing the final system configuration derived in the case where $\alpha = 8$.

When $\alpha = 2$, the output result (Figure 3.16(a)) indicates that we can achieve the required configuration in just one step by transferring Guest_OS_B1 from VMM_B to VMM_A. However when $\alpha = 4$, the output (Figure 3.16(b)) indicates that we need two steps to transfer both Guest_OS_A1 and Guest_OS_A2 to VMM_B to achieve the goal conditions, since the three pieces of software cannot be consolidated into Server_A due

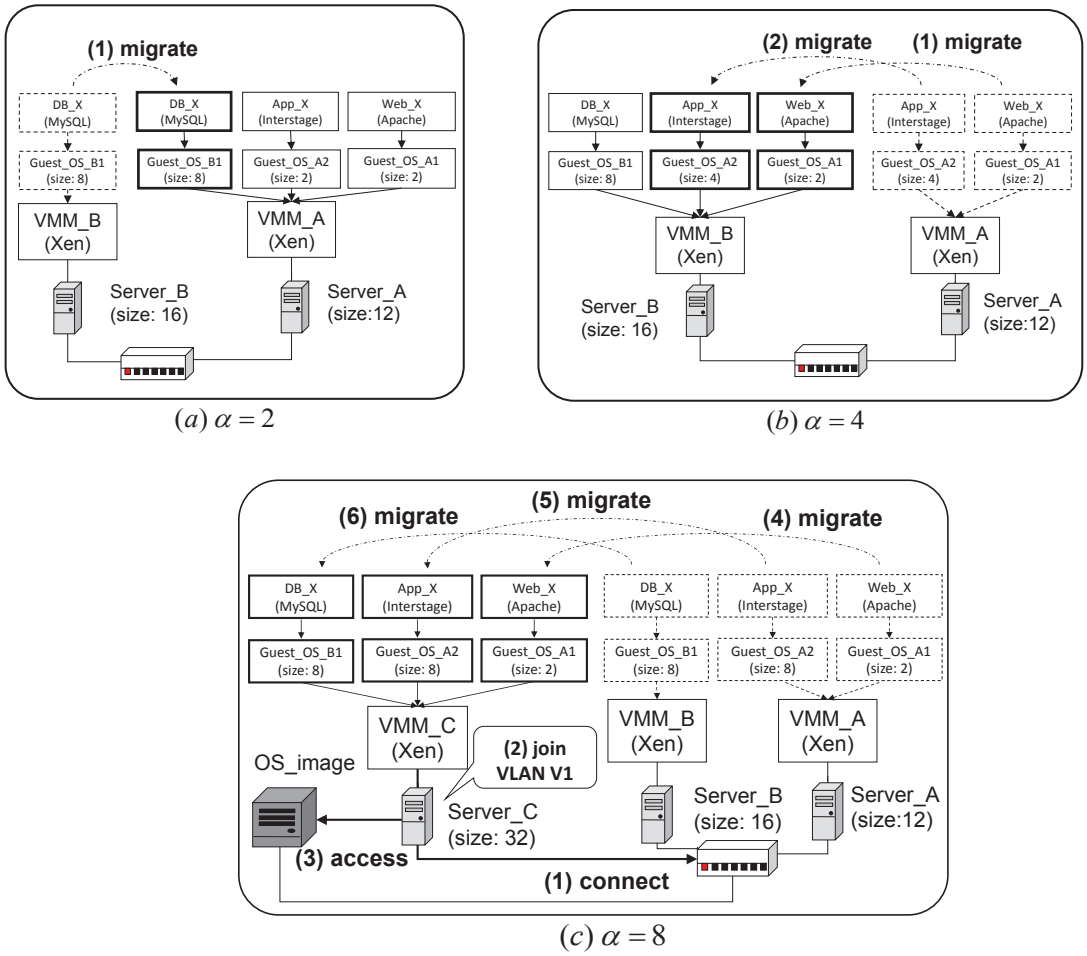


Figure 3.16: Configuration changes planned when size α of Guest_OS_A2 is 2, 4, and 8.

to the capacity constraint. Then when $\alpha = 8$, the procedure consisting of the following six-step operations is obtained (Figure 3.16(c)).

(1) Connect Server_C with Switch_S1.

(2) Incorporate Server_C into VLAN V1.

(3) Establish access from Server_C to OS_image.

(4) Transfer Guest_OS_A1 to VMM_C.

(5) Transfer Guest_OS_A2 to VMM_C.

(6) Transfer Guest_OS_B1 to VMM_C.

This output procedure reveals the following facts. First, due to the capacity constraint, the three pieces of software need to be consolidated on Server_C. However, in order to

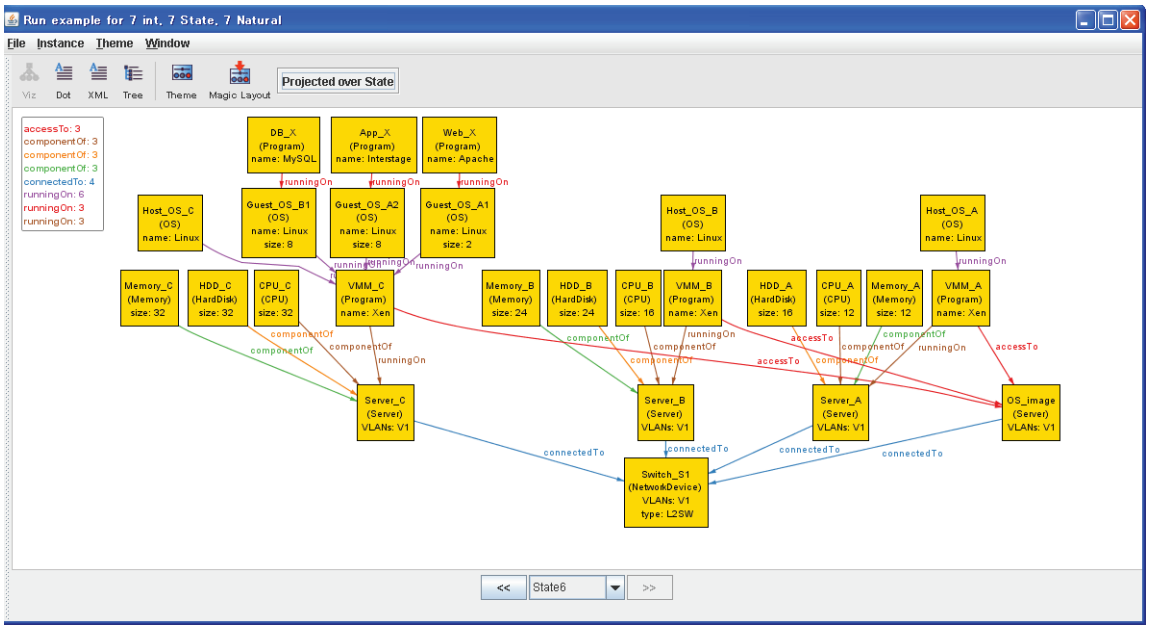


Figure 3.17: Screenshot of planned system configurations when size of Guest_OS_A2 is 8

migrate OSs to Server_C, it should be able to access OS_image. In addition, accessing OS_image requires that Server_C and OS_image belong to the same VLAN segment and a physical connection should be established between them. Therefore, steps 1 to 3 are needed to prepare for the migration to Server_C.

As demonstrated by the above case study, the appropriate procedures for slightly different initial system configurations to achieve the same goal conditions can result in completely different procedures. While these kinds of non-linear and non-straightforward patterns are the essential characteristics of complex systems consisting of various types of components, it is difficult for system administrators to design appropriate procedures without overlooking necessary domain knowledge and the small but critical difference in initial system configurations. Inappropriate procedures designed by administrators with fractional knowledge tend to cause system failures. Our approach based on formalized knowledge, on the other hand, can synthesize appropriate procedures by taking into account both information on current system configurations and knowledge on all domains (e.g., networks and virtual machines).

3.4 Evaluation of computational resource consumption

To evaluate the computational resource consumption of the proposed approach in the procedure synthesis, we compared our approach with the straightforward approach. In this comparison, we set the number of spare servers having the same configuration as Server_C from 1 to 5 in the case study scenario with $\alpha = 8$ (the size of GUEST_OS_A2). Since each spare server is comprised of six components (the server itself, hard disk, memory, CPU,

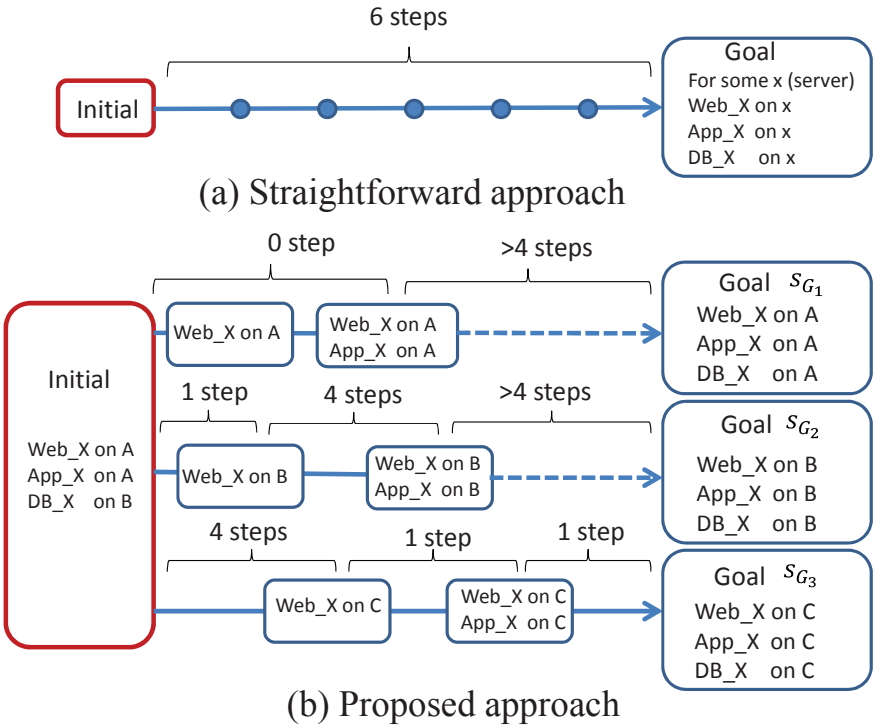


Figure 3.18: State search for case study scenario

virtual machine, and its host OS), the number of components ranges from 26 to 50 in steps of six. We also set the memory size for Alloy Analyzer to 256Mbyte, 512Mbyte and 1024Mbyte and measured the number of SAT clauses for model finding and how much memory is required to synthesize procedures by the both approaches.

In the straightforward approach, we synthesized the whole six steps from the initial state to the goal states using Alloy Analyzer by incrementing the upper limit of the state search one by one until finding the solution, as shown in Figure 3.18(a) On the other hand, our approach only have to synthesize the short operation steps between intermediate goals which lead to a concrete goal state, resulting in less than five steps in the procedure synthesis for each transition to the goal state s_{G_3} as shown in Figure 3.18(b). In this evaluation, we used the three predicates $Pred$={(x in Web_X.^(s.runningOn)), (x in App_X.^(s.runningOn)), (x in DB_X.^(s.runningOn))} to be satisfied in the goal state and constructed the intermediate states along with the order of the predicates in $Pred$. In this case study, since there is no pre-/post-condition relation between these predicates, the order of these predicates in $Pred$ does not affect the resource consumptions and the length of the path representing the synthesized procedure from the initial condition to the goal condition.

From Table 3.1 showing the experimental results, we can see that our approach reduced a third of the maximum number of SAT clauses to be analyzed in the procedure synthesis,

Table 3.1: Evaluation results for SAT clauses and memory size

Number of spare servers	Number of SAT clauses (maximum)			Memory size (byte)	
	Straight-forward	Proposed approach	(ratio)	Straight-forward	Proposed approach
1	2.41×10^6	1.63×10^6	67.4%	512M	512M
2	3.90×10^6	2.64×10^6	67.6%	512M	512M
3	5.92×10^6	4.00×10^6	67.6%	1024M	512M
4	N/A	5.94×10^6	N/A	N/A	1024M
5	N/A	8.51×10^6	N/A	N/A	1024M

comparing with the straightforward approach. It can be also said that the memory size required to synthesize procedures by the proposed approach is less than that with the straightforward approach ("N/A" means non available because of memory overflow or obtaining no analysis results within one hour). Therefore, we can conclude that proposed approach contributed to reduce the amount of resources required to procedure synthesis by reducing the number of SAT clauses.

As for the computational time, the straightforward approach took 68.6 sec in the initial case study scenario with one spare server, while the proposed approach took 76.7 sec. Since multiple paths have to be searched in our approach as shown in Figure 3.18(b), it is possible that the straightforward approach achieves better performance in computational time than the proposed approach. But the difference between them is marginal (about 12% increase).

3.5 Discussion

We demonstrated that our method can synthesize system configuration procedures from the management knowledge regarding operations, constraints and goal conditions. In the case study scenario, we confirmed that we can derive different procedures and final configurations satisfying given goal conditions properly for different initial configurations. In addition, proposed algorithm contributes to reduce the number of SAT clauses to be analyzed in the procedure synthesis. While it has promising potential to improve system management efficiency, we suppose that there are some difficulties for our method to be more practical.

(1) Knowledge management
We can collect various knowledge regarding system management from different stakeholders (e.g. administrators of server, network, database and services). We can easily imagine that there can be some mistakes in described constraints. In addition, the knowledge should be renewed along with the update of the system infrastructure. This kind of flawed or obsolete knowledge can end up with the contradictions between defined knowledge. Since any proper procedure cannot be synthesized from contradicted conditions, proper updates and maintenances of the system management knowledge are quite important.

(2) Efficient modeling
As shown in the Section 3.4, we reduced the number of SAT clauses to be analyzed

for procedure synthesis by using intermediate goal states. However, in order to analyze larger system, we need to make the number of SAT clauses as small as possible. There will be some approached to achieve this. For example, we can apply some effective CNF construction techniques proposed in [35]. It will also be possible to reduce the size of system model by abstracting some parts irrelevant to the purpose of system configuration changes.

3.6 Summary

In order to design proper configuration change procedure in a private cloud system consisting of various types of components, it is needed to use every expertises regarging the components. To solve this burdensome problem, we developed a method of synthesizing procedures for system configurations using the Alloy Analyzer model finder. In this method, first, management knowledge such as executable operations, constraints, and goal conditions are described in the Alloy language. Next, this knowledge is combined with information on current system configurations derived from CMDB. Then the Alloy Analyzer identifies a model satisfying all conditions from integrated information. We can regard the model as an appropriate procedure (a sequence of operations) that can change the system from its initial configuration to the goal configuration that satisfies given constraints. In the procedure synthesis, we divide the possible paths from initial states to concrete goal states into smaller steps using intermediate goal states to reduce the memory resources required in the synthesis. Through a case study with typical scenario of virtual machine consolidation for energy savings in cloud infrastructure, we confirmed that our method could derive appropriate procedures for different configurations and evaluated the memory consumptions.

Since our method will enable us to obtain appropriate procedures for configurations automatically without forgetting to take into account the necessary knowledge on system management, the reliability of system management can be improved and we can reduce the cost needed to check the validity of obtained procedures and fix any problems caused by inappropriate procedures constructed by human experts. Since it can also handle segmented fractional knowledge on system management possessed by individual domain experts and information on system configurations in the same declarative manner, it will make it easy to maintain (add, remove, or revise) knowledge on system management. These characteristics are crucial for breaking down "silos" in private cloud system management, where management knowledge possessed by domain experts needs to be frequently modified in line with the changing requirements of systems.

We are now considering the following work for the future. First, while we can flexibly define management knowledge in the Alloy description based on first-order logical formulas and set theory, it might take some time for system administrators to learn how to describe their knowledge in the Alloy language. To overcome this problem, we are going to prepare various interfaces to help them to easily describe their knowledge. For example, it would be helpful if we had some format for knowledge description and a translation function that could automatically translate the format filled in by domain experts into Alloy.

Next, reducing computational complexity is also an important goal for future work. While our algorithm using intermediate goals contributed to reduce the size of SAT formulas, we consider that there are still rooms for improvement in simplifying system models.

Combining techniques for making simple model with a model finding technique that can check the satisfiabilities of formulas consisting of over 10 million SAT clauses, it would be possible to overcome scalability problems in today's complex large-scale ICT systems.

Chapter 4

Operational vulnerability evaluation

4.1 Problem: Misconfiguration in redundant structure

We discussed the method for configuration change procedure synthesis in the previous chapter. This method is aimed at improving the reliability of system management by avoiding misconfiguration in the design phase of configuration changes. However, there still are rooms for misconfigurations causing service failures in the execution phase of the configuration changes. For example, suppose redundant structures (e.g., load balancing and high availability clusters) which are commonly applied to cloud systems and other information systems to prevent service failure. System administrators assume that the redundant structures can prevent a single hardware fault in their infrastructure from progressing to a service failure that is discernible to users. However, there remain many cases where the redundant systems are disabled by misconfigurations or improper operation executions. For example, if several virtual machines for redundancy are instantiated on the same physical server, it is possible that a failure in the physical server can make all of these virtual machines inactive. In such cases, the flaws hidden in the system become apparent only after the occurrence of service faults triggered by events such as hardware faults. To prevent such disastrous situations, we need to identify the weak points (e.g., single points of failure) lurking in the system configurations, and the types of operations that can cause serious service failures. In other words, to realize reliable cloud infrastructure management, we need to evaluate the "vulnerability" of systems to accidental events in the cloud system management lifecycle, such as component faults and improper operations. By identifying and eliminating the hidden risks in the systems that are associated with their operation, we can improve the quality and the reliability of cloud service management. From the information presented above, we propose a framework to evaluate the vulnerability of systems to internal system operation factors such as component faults or misconfigurations. In this framework, we first define the vulnerability level of services based on the degree of failure impact caused by misconfigurations or component failures. Next, we construct a model representing the behavior of a cloud system by combining models for components comprising the cloud system with interdependency relationships between the components. In this model, we represent the execution of configuration change operations and the occurrence of a component fault as a state transition in a component model. A state change in a component can trigger state changes in an-

other component due to the interdependency between these components. By using this model, we enable the analysis of the propagation of the fault's effect between components having interdependencies. Then, we use a case study to demonstrate how to evaluate the operational vulnerability of a service using the model checker NuSMV.

The main contributions of this chapter can be summarized by the following three points: (1) we defined the operational vulnerability level for cloud services, (2) we presented a method for the construction of a cloud system model and its representation in a language for model checking, and (3) we demonstrated the evaluation using a model checking tool.

The rest of this chapter is organized as follows. In Section 4.2, we first define the types of systems and management operations to be discussed in this chapter. Next, we define the degree of operational vulnerability. In Section 4.3, we show how to construct a state transition model that represents the system's behavior. In Section 4.4, we use this model to demonstrate the evaluation of s system's vulnerability by carrying out a case study using the model checker NuSMV. Then, we discuss our results in Section 4.5. Finally Section 4.6 concludes the chapter and outlines future challenges.

4.2 System, operation and vulnerability

4.2.1 Target systems to be analyzed by our framework

In this chapter, we analyze the behavior and the vulnerability of services instantiated on IaaS (Infrastructure as a Service)-type cloud computing infrastructures with virtualization technologies. Cloud providers with IaaS architectures have physical servers and network components (e.g., routers and switches), and use virtualization functions called Hypervisor such as Xen [44] and VMware [45] to instantiate several virtual machines (VMs) on a physical server. They lend these VMs as computing resources to their customers and charge rental fees in a pay-per-use manner for the resources (e.g. CPU time, storage usage and amount of data transmitted). Typical examples of these IaaS services include Amazon EC2 [33] and Google Compute Engine [46]. The other types of cloud services are PaaS (Platform as a Service), which provides environments for software development and deployment, and SaaS (Software as a Service), which provides software functions as services; however, these types of cloud services are out of the scope of this chapter.

Using the IaaS cloud infrastructure, customers of cloud services can construct various types of systems to realize their specific services. One of the typical system configurations of enterprise information systems is a 3-tier system consisting of three types of functions: (1) web tier, which provides a frontend interface for users of services, (2) application tier, which handles business logics and (3) database tier, which is responsible for the management of stored data. Actually, some cloud providers provide customized functions to make it easy to construct virtualized multi-tier systems on their cloud infrastructure [47]. Based on the above explanation, we selected as our target for the vulnerability evaluation a service that is realized by a virtualized multi-tier system instantiated on an IaaS cloud infrastructure.

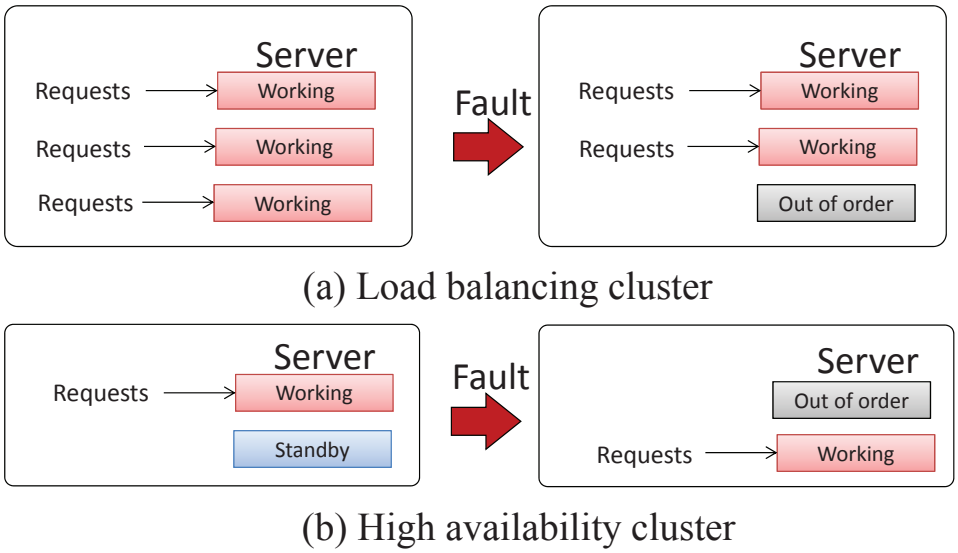


Figure 4.1: Cluster structure for redundancy

4.2.2 Type of failures and operations

With respect to the failures occurring in the system, we only consider crash faults in components (e.g., hardware, software and VM) comprising the cloud system. We do not take into account the other faults such as partial and temporal faults (e.g., transient fault, intermittent fault and omission fault), faults related to performance (e.g., timing fault and response fault) and faults for which we cannot predict the effects (e.g., Byzantine fault).

With regards to the type of operations that can be executed in the system, we consider two types of operations. One is to change the state of components and the other is to change the interdependencies between components. The former includes operations such as "server shutdown," which changes the server's power status from "On" to "Off." The latter includes operations such as live migration, which transfers a virtual machine between physical servers.

4.2.3 Operational vulnerability

The operational vulnerability discussed in this chapter indicates how susceptible services are to accidental and undesirable internal events such as component faults and improper operation executions. Because we consider only internal accidental factors related to cloud infrastructure management, we do not consider the vulnerability from the perspective of system security, that is, the resistance of services and systems to intentional and malicious attacks external to the system.

Usually, in the multi-tier systems described in Section 4.2.1, some redundancy structures are applied to the systems so that a hardware fault does not lead to service failures discernible to users [48]. The following typical patterns are shown in Figure 4.1.

(a) Load balancing cluster
This distributes user requests for the service to several components running at the

same time. In this configuration, when a component fails, the rest can continue the service. A typical example of this configuration is a load balancing function for frontend web servers.

(b) High availability (HA) cluster
This keeps some components as backups for a function in case there is a fault for the primary component enabling the function. When the primary component is working properly, the backup components are on stand-by and monitor the status of the primary component. When the primary component fails, a backup starts up and substitutes for the primary. This structure is used when it is difficult to apply the load balancing cluster. For example, since a database must maintain the consistency of its data, it is sometimes difficult to apply a distributed database server structure that can require several database servers to simultaneously access the same data. In such a case, we can use the HA cluster structure in which we connect a primary and a backup database server with shared storage servers. When the primary database fails, we can continue the service and maintain the data consistency by starting up the backup database server immediately.

However, even if we use these redundant structures, service failures can occur because of reasons such as component faults and improper operations. By focusing on the number of components comprising a function and a service, we can categorize the service failures into the following two types.

(a) Lack of working components
A system cannot continue to provide its services when there are not enough components to provide a function necessary for the service. For example, even if we use 10 Web servers with a load balancing structure in a 3-tier system, the service will fail when all of these 10 Web servers fail. This type of service failure is solved by just restarting or substituting the failed functions.

(b) Excess of working components
A service failure can also occur when the number of components providing a function exceeds its limit. For example, for database servers in an HA cluster structure, we need to keep the system so that only one server (either a primary database or a backup) in the cluster is working. However, if we have misconfigurations in which the backup server that is supposed to monitor the primary database is configured to monitor a different component, these two database servers might become active at the same time because a fault in the monitored component can trigger the starting up of the backup server while the primary one is still working. This case can result in a "split brain" failure [48], which causes inconsistencies in the data accessed from different database instances at the same time. To recover from this type of failure, we need to not only reduce the number of working database server instances, but also to execute the recovery procedures for the data having inconsistency, which can be burdensome and time-consuming. In addition, if the cloud providers cannot recover their customers' data, the impact of the failure is immeasurable. Therefore, this type of failure has a larger impact on the service than the case involving a lack of working components.

Based on the above categorization, we defined the operational vulnerability of services from level 0 to 3 as follows.

- **Level 0: Safe condition**

  This is a scenario in which an infrastructure is configured properly and a service is working properly. In this level, one component fault or improper operation does not invoke a service failure that can be perceived by users.

- **Level 1: Single point of failure**

  This is a scenario in which a service is working, but one component fault or improper operation can lead the system into the undesirable scenarios defined by level 2 and level 3 below.

- **Level 2: Service unavailable**

  This is a scenario in which a service is not working because the components comprising a function are not alive. We can resume the service by restarting one of the failed components.

- **Level 3: Data inconsistency**

  This is a scenario in which the number of components enabling a service exceeds its limit. Data loss or data inconsistency can occur in this scenario. This scenario is more serious than that in level 2 because the recovery of the damaged data will take time and incur additional cost. In this chapter, we do not take the types of impaired consistencies (e.g., storing consistency and eventual consistency) in database into account.

Our main purpose in this chapter is to evaluate the operational vulnerability of a system by determining the vulnerability level in which the system exists. However, it is difficult to evaluate the vulnerability using simple approaches such as an analysis of the static system configuration (snapshot at a certain instant). For example, a multi-tier system instantiated on a cloud infrastructure consists of various components such as virtual machines and physical servers. These components collaborate and interrelate with each other to provide a service in an organized way. For example, if a physical server crashes, virtual machines instantiated on the server also shut off. If a primary virtual machine in an HA cluster shuts off, the backup virtual machine monitoring the primary one can be activated. From these examples, we see that a state change in a component can affect other components. In addition, the interdependency between components can be changed by some configuration change operations. Therefore, in order to evaluate the operational vulnerability, we need to construct a model that can represent the state changes in components and the propagations of the state changes along with the interrelationship between these components. Details of the construction and analysis of such models are explained in the next section.

4.3 Construction of system model and property

In this section, we explain how to construct system models and properties to be verified, which are essential factors in our evaluation of operational vulnerability. First, to

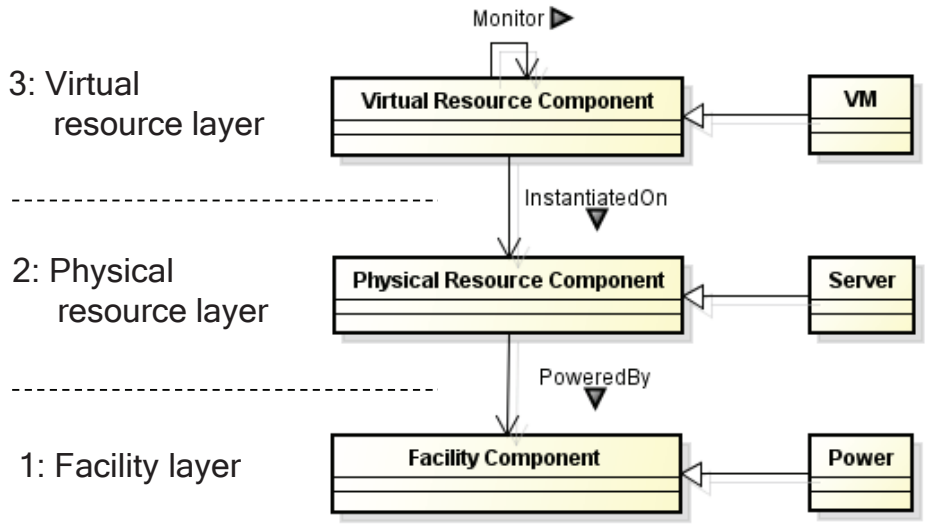


Figure 4.2: 3-layer system structure model

construct the system model, we define layers of components comprising a system and interdependencies between the components. Next, we use a state transition model to represent the status changes in each component caused by component faults or configuration change operations. In this model, the state transition of a component can trigger another state transition in another component according to the interdependency between them. Then, we explain how to construct properties to be checked in order to conduct an evaluation of the operational vulnerability of a system.

4.3.1 Layers of components and interdependencies

In this chapter, we consider as our analysis target a virtualized cloud system that can lease computing resources in units of a virtual machine to customers. The virtual machines are instantiated on physical servers. In datacenters, facility components such as power supplies provide electric power to enable the physical servers to work. To construct a system model having such a structure, we defined a 3-layer system model having a structure depicted in Figure 4.2. The detailed explanation for each layer in this figure is as follows.

(1) Facility layer
This is the layer that represents facilities to support physical computing resources, and include components such as power units, which provide electric power to physical servers.

(2) Physical resource layer
This is the layer consisting of physical components accommodating virtual resources to be leased to cloud users, and includes components such as physical servers on which virtual machines are instantiated.

(3) Virtual resource layer
This layer consists of virtualized computing resources to be leased to customers of cloud providers, and virtual machines on which operating systems are installed belong to this layer.

While we have depicted only three types of components (power unit, physical server and virtual machine (VM)) in Figure 4.2 for simplicity, the other types of components involved in the provision of services can be categorized into one of these layers. For example, a physical storage server can belong to the physical resource layer since it is a physical hardware component that stores virtual storages used by customers.

We also define the types of interrelationships between these components as follows. These interrelationships are used to represent the propagations of state transitions between components in the system model.

(1) PoweredBy relation
An interrelationship between two components where a component in the facility layer provides electric power to the other component in the physical layer. No component in the physical layer can work without a power supply.

(2) InstantiatedOn relation
An interrelationship between two components where a component in the virtual resource layer is instantiated on the other component in the physical layer. No component in the virtualized layer can work without being instantiated on a component in the physical layer.

(3) Monitor relation
An interrelationship between two components in the virtual resource layer where one component monitors the behavior of the other component. When a monitored component changes its status, the other component monitoring it can also change the status. The interrelationship is used to represent the relation between the primary and the backup component in a HA cluster structure where the fault of the primary component triggers the activation of the backup component.

4.3.2 State transition in a single component

In order to construct the system model representing the behavior of the entire system providing a service, we first construct a state transition model representing the behavior of a single component in the system.

Definition 1 (States of a component) *Let $C_{Fac}, C_{Phy}, C_{Vir}$ be the set of components belonging to the facility layer, the physical resource layer and the virtual resource layer, respectively. We define the set of states of a component $c \in C_{Fac} \cup C_{Phy} \cup C_{Vir}$ in the system by $S_c = Status(c) \times Pow(c) \times Ins(c) \times Mon(c)$.*

$Status(c) = \{StandBy, On, Off\}$ represents the power status of c ; On (working properly), Off (not working) and StandBy (waiting for starting up). The set $Pow(c) = 2^{C_{Fac}}$ represents the set of components belonging to the facility layer that have a PoweredBy relation with the component $c \in C_{Phy}$.

Likewise, the set $Ins(c) \subseteq 2^{C_{Phy}}$ represents the set of components in the physical resource layer having an InstantiatedOn relation with the component $c \in C_{Vir}$. Note that

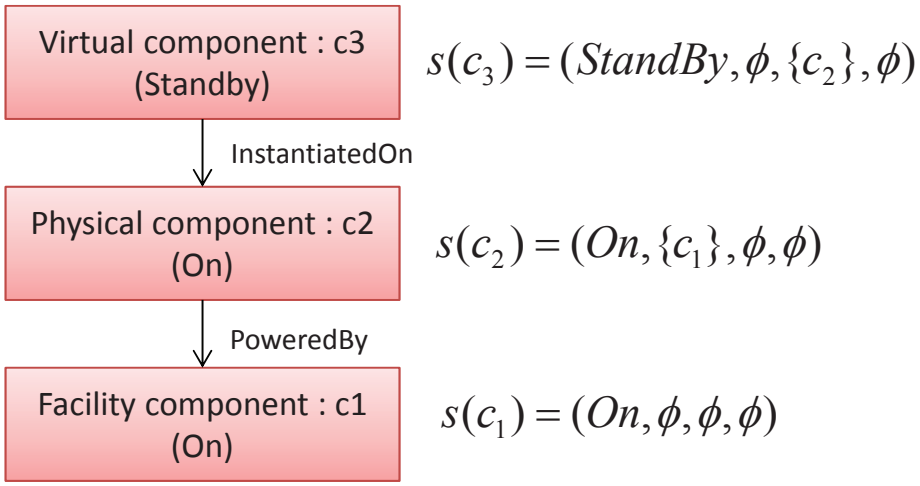


Figure 4.3: Example of a system model consisting of components and their interdependency

$Ins(c)$ is a singleton set or empty set because the virtualized component c can be instantiated on only one physical component at a certain instant. $Mon(c) = 2^{C_{Vir}}$ represents the set of components having the monitor relation with $c \in C_{Vir}$.

For example, the states of three components c_1, c_2, c_3 in Figure 4.3 can be represented by $s(c_1) \in S_{c_1}$, $s(c_2) \in S_{c_2}$ and $s(c_3) \in S_{c_3}$, respectively.

By using the state definition, we define the state transitions $T = S \times S$ triggered by the component fault or configuration change operations as follows. Here, we only consider two types of operations: (1) live migration [49], which involves transferring a virtual machine from a physical server to another server, and (2) changes in the monitoring target in the HA cluster structure.

Definition 2 (State transition by component failure)

$$\begin{aligned} T_{Fault} \quad = \quad & \{(s(c), s'(c)) | (s(c) = (StandBy, x, y, z) \\ & || s(c) = (On, x, y, z)), s'(c) = (Off, x, y, z), \\ & c \in C_{Fac} \cup C_{Phy} \cup C_{Vir}\} \end{aligned}$$

$T_{Fault} \subseteq T$ represents the state transition triggered by the occurrence of a fault in the component c. The fault invokes the state transition from s (the power status is On or StandBy) to s' (the power status is Off).

Definition 3 (State transition by live migration)

$$\begin{aligned} T_{Migration} \quad = \quad & \{(s(c), s'(c)) | s(c) = (w, x, y, z), s'(c) = (w, x, y', z), \\ & y \neq y', c \in C_{Vir}, y \in C_{Phy}, y' \in C_{Phy}\} \end{aligned}$$

$T_{Migration} \subseteq T$ represents the transition triggered by the execution of live migration, invoking the state transition from s (a virtual machine c is instantiated on a physical server y) to s' (c is on server y').

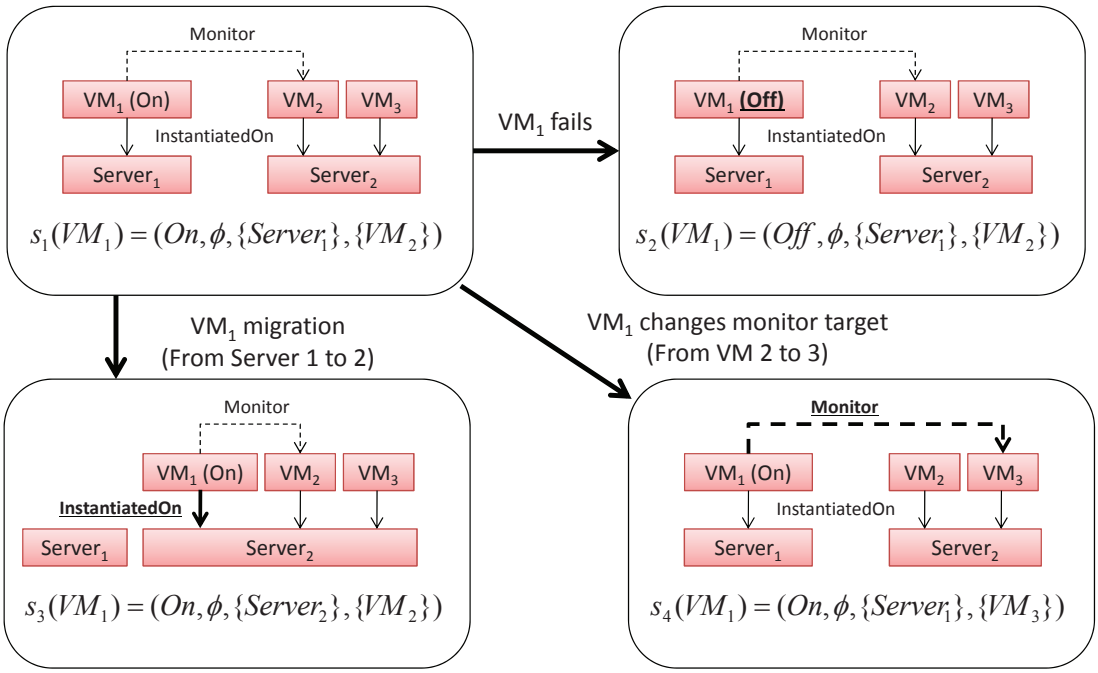


Figure 4.4: State transition by faults and operations

Definition 4 (State transition by changing monitor target)

$$T_{Change\_monitor} = \{(s(c), s'(c)) | s(c) = (w, x, y, z), s'(c) = (w, x, y, z'), z \neq z', c \in C_{Vir}, z \in C_{Vir}, z' \in C_{Vir}\}$$

$T_{Change\_monitor} \subseteq T$ represents the transition triggered by the operation changing the monitor target of a virtual machine c from z to z'.

For example, when a component VM_1 is in a state s_1 depicted in Figure 4.4, a fault in VM_1, the live migration of VM_1 and changing VM_1's monitored target can invoke the state transition from s_1 to s_2, s_3, and s_4, respectively.

4.3.3 State transitions propagating along with relations

Here, we define the state transitions in a component triggered by the state transitions that occur in other components, along with the interrelationships between these components. Here, we suppose the following two types of propagations of state transitions: (1) propagation of power shut down and (2) starting up from standby by detecting the powering off of other components.

Definition 5 (Propagation of power shut down)

$$T_{Propagate} = \{(s(c), s'(c)) | s(c) = (w, x, y, z), s'(c) = (Off, x, y, z), Pow(c') = Off, (c' \in x || c' \in y)\}$$

$T_{Propagate} \subseteq T$ represents the state transitions in a component c becoming Off when a component c' shuts off, and c depends on c' by PoweredBy or InstantiatedOn relations. This transition is used to represent situations where a physical server is shut down because power units that provide electric power to the physical server shut down. Likewise, if a physical server shuts down, all of the virtual machines that are instantiated on it will also shut down.

Definition 6 (Starting up from standby mode)

$$\begin{aligned} T_{Wakeup} &= \{(s(c), s'(c)) | s(c) = (StandBy, x, y, z), s'(c) = (On, x, y, z), \\ &\quad Pow(c') = Off, c \in C_{Vir}, c' \in z\} \end{aligned}$$

$T_{Wakeup} \subseteq T$ represents the state transitions where a backup component c that is monitoring a primary component c' changes its power status from StandBy to On when c detects that c' is not working.

For example, suppose that a system consisting of one physical server ($Server_1$) and two virtual machines (VM_1 and VM_2), as depicted in Figure 4.5, is in its initial state s_1. If $Server_1$ fails, the system changes its state to s_2, where $Server_1$'s power is Off. Because VM_1 has an InstantiatedOn relation with $Server_1$, the system's state changes to s_3, where VM_1 is also Off through the transition $T_{Propagate}$. Likewise, if VM_2 fails in the initial state s_1, the system changes its state to s_4. Then, it moves to s_5, where VM_1 monitoring VM_2 starts up after detecting VM_2's shutdown.

4.3.4 Property description

In order to evaluate the operational vulnerability of services, we need to describe properties that should be satisfied in the system that works properly using logical formula. We define the properties to be checked for the vulnerability of level 2 and 3 described in Section 4.2.3 as follows.

Definition 7 (Property for vulnerability level 2 (service unavailable))

$$\begin{aligned} &AG(AF(Service\_running)), \\ &Service\_running := (func_1 \& func_2 \& ... \& func_n), \\ &func_i = (Status(c_{i,1}) = On | Status(c_{i,2}) = On | ... | Status(c_{i,m_i}) = On) \end{aligned}$$

Here, we suppose that a service consists of n functions, and the i-th function consists of m_i redundant components. In this case, in order for the service consisting of these functions to be working, at least one component should be alive for each function. The temporal logical formula $AG(AF(Service\_running))$ described in CTL (Computational Tree Logic) represents this condition. The proposition $Service\_running$ consists of the propositions $func_i$ representing each function. From the AND-concatenation of $func_i$, it represents that all functions should be alive for the service to be available. Likewise, $func_i$ consists of the OR-concatenation of the proposition $state(c_{i,j}) = On$, indicating that at least one component should be alive in the i-th function. The temporal operators AG

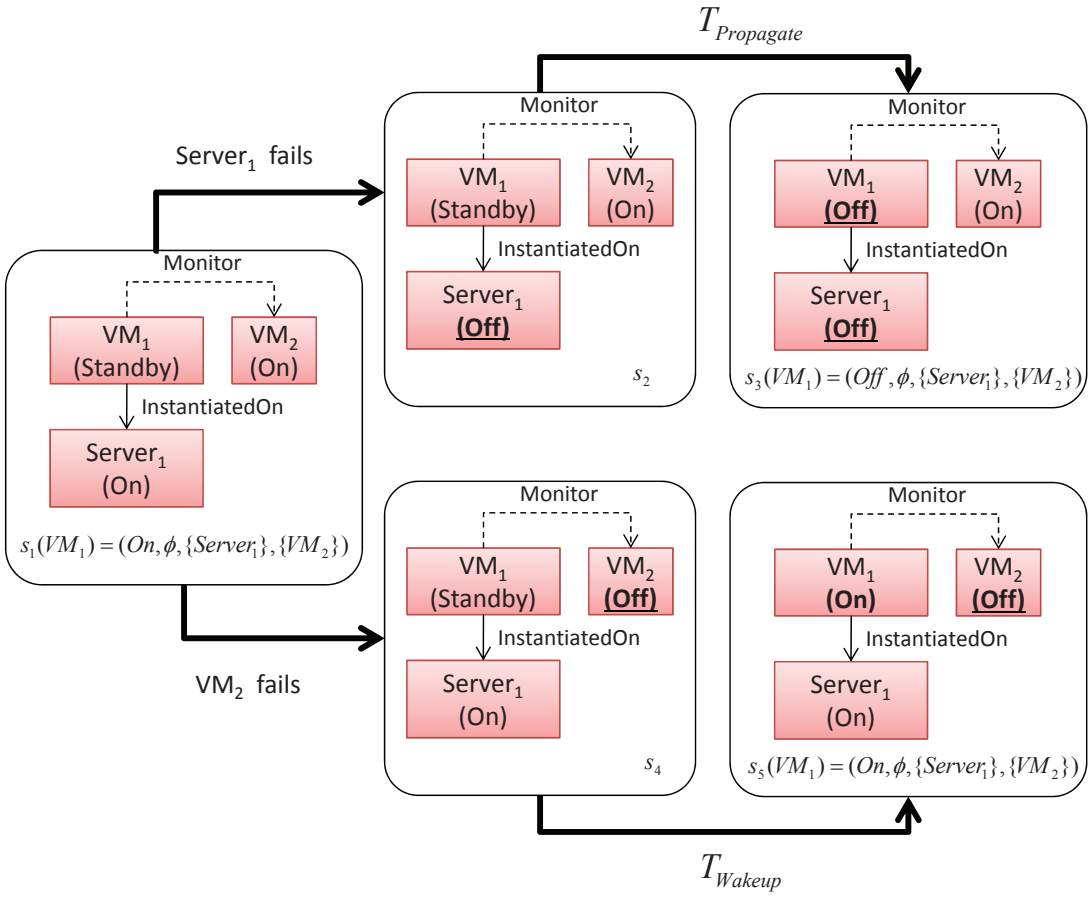


Figure 4.5: State transition by propagation of state changes

and AF represent "always globally" and "always finally" respectively. $AG(X)$ indicates that X is always satisfied. $AF(X)$ indicates that X is eventually satisfied. Therefore, $AG(AF(Service\_running))$ indicates that even if the service is temporarily unavailable, it will eventually resume. For example, suppose that VM_1 and VM_2 in Figure 4.5 comprise a function in a system. If VM_2 fails in the initial state s_1, the system moves to the next state s_4. In the state s_4, neither VM_1 nor VM_2 is functioning. While the proposition Service_running is not satisfied at this instant, the backup component (VM_2) of the HA cluster wakes up immediately and the system changes its state to s_5. As shown in this figure, if the system can recover its all functions under some given conditions (e.g. the number of simultanerous component faults is less than 2), the model satisfies the property. Otherwise, since it might not be able to recover some functions, we can conclude that the system is in the level 2 vulnerability.

Definition 8 (Property of vulnerability level 3 (data inconsistency))

$$AG(\neg SplitBrain),$$
$$SplitBrain := (Sb_1|Sb_2|...|Sb_n),$$
$$Sb_i := (Ex_{i,1}|Ex_{i,2}|...|Ex_{i,m_i}),$$
$$Ex_{i,j} := (Status(ex_{i,j}) = On\&(Status(ex_{i,1}) = On|Status(ex_{i,2}) = On|...|$$

$$Status(ex_{i,j-1}) = On | Status(ex_{i,j+1}) = On | ... | Status(ex_{i,m_i}) = On)$$

Here, we suppose that a service consists of n functions, and the i-th function has m_i components that are supposed to work exclusively. Therefore, to avoid data inconsistency caused by a split brain, only one of m_i components is allowed to be active in the i-th function. The proposition $Ex_{i,j}$ indicates that the j-th component and one other component are active in the i-th function. By concatenating $Ex_{i,j}$ with OR-operations, Sb_i indicates that at least two components are active in the i-th function, meaning that a split brain is occurring. By the OR-concatenation of Sb_i, the proposition SplitBrain means that a split brain occurs in some functions. Therefore, the logical formula $AG(\neg SplitBrain)$ means that a split brain never occurs in any situation. A system not in the vulnerability level 3 should satisfy the property.

We can determine that a system is in level 2 or 3 of operational vulnerability if the model representing the behavior of the system does not satisfy these properties. If we inject one fault in a component to the system model, and it does not satisfy these properties, it means that the system has a "single point of failure", and its operational vulnerability level is 1. If we have to inject more than one fault into the system to satisfy these properties, we can conclude that the vulnerability level is zero since the system has no single point of failure.

4.4 Demonstration of vulnerability evaluation

In this section, we demonstrate the evaluation of the operational vulnerability of a service instantiated on the cloud system through a case study scenario. In this demonstration, we use a model checking approach to showing how to verify properties for the state transition model representing the behavior of the cloud system. The reason that we decided to use the model checking for the operational vulnerability evaluation is that its capability to execute the exhaustive search is suitable for checking the effect of a fault and operations without overlooking vulnerabilities hidden in a system. In our approach, the cloud system model presented in the previous section consists of components having several discrete states. The behavior of the system is determined by the interaction between these components. Common tools for the verification of such a distributed system include SPIN [52], PRISM [53] and NuSMV [54]. We decided to use NuSMV, because it is one of the most popular open-source model checking tools which can verify CTL (Computational Tree Logic) formula in which we describe the properties to be checked.

4.4.1 Translation from state model to NuSMV code

In our approach, we translate a system model presented in the previous section to NuSMV code as follows.

Components in the system

We define each component comprising the system as a module in NuSMV. In this module, we define the following three types of state transitions regarding the power status of the component.

```
MODULE VM(no, State, fail, inston, mon)
ASSIGN
next(State[no]):= case
fail[no]=1: Off;
State[inston]= Off: Off;
State[no] = StandBy & State[mon] = Off: On;
TRUE: State[no];
esac;
```

Figure 4.6: Definition of VM component

(1) State transition triggered by a fault of the component itself.

(2) Propagation of shutting off between components via the `PoweredBy` and `InstantiatedOn` interrelationship.

(3) Starting up from the StandBy mode via the `Monitor` interrelationship when a monitored component shuts off.

For example, the state transitions in a module representing the behavior of a virtual machine component that is monitoring another component can be defined as shown in Figure 4.6.

The above NuSMV code defines the value of the power status (`State`) of a component using its identification number (`no`) in the next state by the following conditional state transitions.

(1) Component fault: If the component fails, it shuts off. (`fail[no]=1: Off;`)

(2) Propagation of shutting down: If a physical server `inston` having `InstantiatedOn` relation with the component fails, the component shuts off. (`State[inston]= Off: Off;`)

(3) Starting up from StandBy: If the component in the standby mode and another component `mon` that is being monitored by it fails, it starts up. (`State[no] = StandBy & State[mon] = Off: On;`)

(4) Others: Keep the current state (`TRUE: State[no];`)

The components in the physical layer and the facility layer can be modeled in the same manner, while they have to have a `PoweredBy` relationship instead of `InstantiatedOn` and `Monitor` relationships. The behaviors of `PoweredBy` and `InstantiatedOn` relationships are the same because in both relationships, if one component shuts off, the other component that relies on it also shuts off.

```
MODULE Fault(i,fail)

DEFINE
f_th   := LIMIT; -- Fault threshold

VAR
rand   : {0,1,2,3,4,5};

ASSIGN
init(rand):=0;
next(rand):= case
f<f_th: {1,2,3,4,5};
TRUE: 0;
esac;

next(fail[ 1]):= case rand = 1 : 1; TRUE: fail[ 1]; esac;
next(fail[ 2]):= case rand = 2 : 1; TRUE: fail[ 2]; esac;
next(fail[ 3]):= case rand = 3 : 1; TRUE: fail[ 3]; esac;
next(fail[ 4]):= case rand = 4 : 1; TRUE: fail[ 4]; esac;
next(fail[ 5]):= case rand = 5 : 1; TRUE: fail[ 5]; esac;
```

Figure 4.7: Model of fault occurrence

Faults and operations occurring in nondeterministic way

In our model, we suppose that faults in components and improper configuration change operations can happen in a nondeterministic way. Therefore, we simulate the occurrence of faults and configuration changes by defining a module which randomly selects a component from the model and invokes a change in the components' power state or interrelationship between the components. For example, using the module definition shown in Figure 4.7, we can define state transitions showing that one of five components can fail nondeterministically.

In this NuSMV code, the line "`f<f_th:  {1,2,3,4,5};`" gives to the variable `rand` a random value from 1 to 5 if the number of faults `f` does not exceed its limit `f_th`. Then, it changes the value of the variable `fail` for the component with identification number `rand` to 1 (true) in order to indicate that a fault occurred in the component. This change invokes the state transition in the module representing the behavior of the component, as shown in the previous subsection. As a result, the component's power state changes to `Off`. Executions of operations such as live migration and changes in the monitor target can be defined in the same way, while we need to choose two components; one is the target of the configuration change operation and the other is the destination of the live migration or the monitored component.

4.4.2 Case study scenario

Here, we demonstrate how to evaluate a system's operational vulnerability using model checking through a case study scenario. This scenario is designed as one example of analysis of vulnerability caused by undesirable events which can occur in cloud user side and cloud provider side. The detail of the events (faults and configuration changes) is

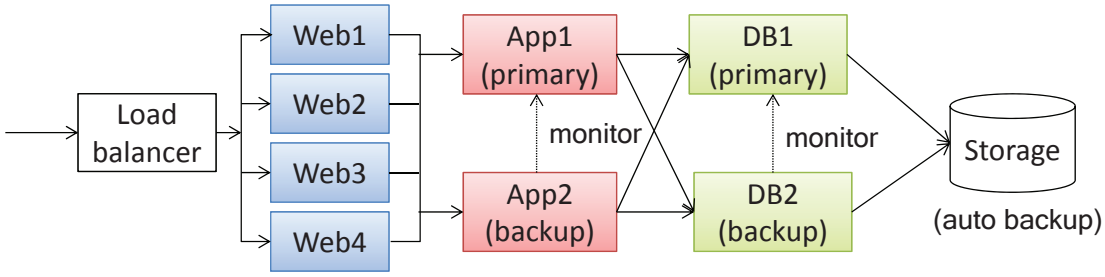


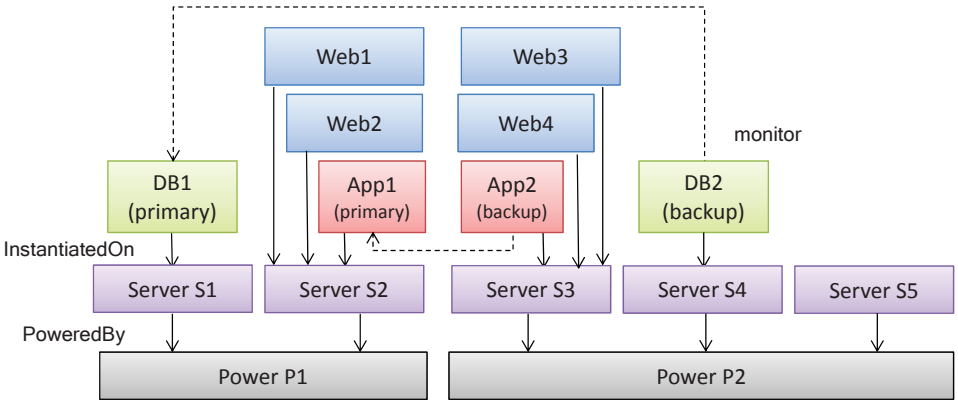
Figure 4.8: 3-tier system for case study scenario

described later. Here we suppose that a cloud user contracting a cloud service provider provides its service to customers by constructing the three-tier system shown in Figure 4.8 using virtual machines on the cloud infrastructure.

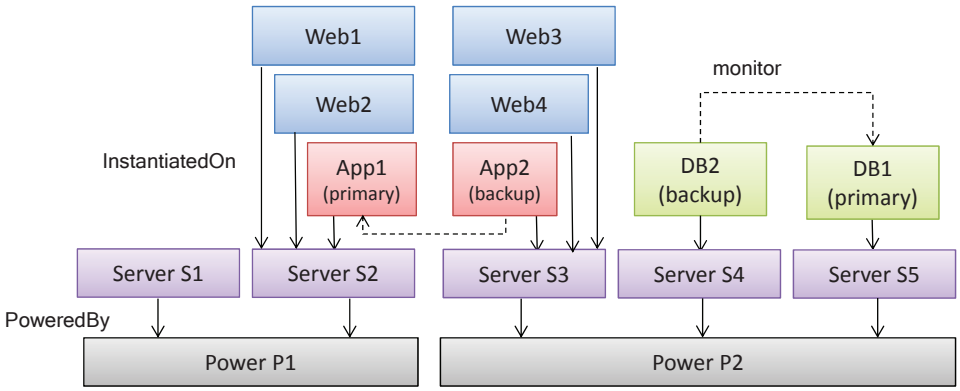
The three-tier system consists of a web server tier, an application server tier and a database server tier. The web tier consists of four virtual machines (Web1, Web2, Web3 and Web4) with a load balancing cluster structure by using a load balancing function (e.g., Amazon EC2 Elastic Load Balancing [55]) provided by the cloud provider. The application layer is the HA cluster having a virtual machine App1 as its primary and App2 as its backup. The database layer has the same structure using two virtual machines (DB1 for primary and DB2 for backup). The data managed by the database servers is stored in a storage prepared by storage services (e.g., Amazon S3 [56]), which is provided by the cloud provider. Here, we assume that the load balancing function and the storage service do not fail because the cloud provider provides these services and prepares a redundant structure for them. Virtualized systems consisting of around 10 servers are common configurations for middle-sized web services. For example, gumi [63], which is a popular social game site in Japan having more than 10 million users, uses 8 virtual machines for web-cum-application servers and 2 virtual machines for database servers [64]. Therefore, it is practical to assume that a system for a typical web service for less than 10 million users can consist of less than 10 virtual machine instances.

The manner in which these virtual machines are distributed on physical servers in the cloud infrastructure depends on the decision made by the cloud provider side. Here, we evaluate the operational vulnerability of the service with the placements in pattern A and B, which are depicted in Figure 4.9(a) and (b), respectively, in order to show the difference between the evaluation results of the operational vulnerability for these similar placements.

In each deployment pattern, the cloud infrastructure consists of two power units (P1 and P2) in the facility layer and five physical servers (S1, S2, S3, S4 and S5) in the physical resource layer. The power unit P1 provides its electric power for S1 and S2, while P2 provides for the other physical servers. In the deployment pattern A, S2 accommodates three virtual machines (two web servers (Web1 and Web2) and one primary application server App1). S3 has two web servers (Web3 and Web4) and one application server App2



(a) Deployment pattern A



(b) Deployment pattern B

Figure 4.9: Deployment patterns of virtual machines

for backup. The virtual machines for the primary database server (DB1) and for the backup (DB2) are deployed on S1 and S4, respectively. In pattern B, it has the same placement as with pattern A, except that DB1 is on S5. In the initial state in both patterns, the virtual machines for backup (App2 and DB2) are in the standby mode, and the other virtual machines are working properly.

In this case study, we assume that the following faults and operations can happen in the system.

Fault: As typical events in information systems, we suppose crash faults of components can occur. For cloud provider side, they include the faults in power units, physical servers and virtual machines caused by hardware/software defects. For cloud user side, they include the events such as application halting and unintentional shutdowns.

Live migration operation: As one of the typical examples of operations executed in cloud provider side, we suppose that live migration can be executed to transfer virtual machines to different physical server. As shown in [57], live migration is one of the common tasks executed in cloud datacenter.

```
AG(AF(Service_running))
Service_running :=
(State[Web1]=On | State[Web2]=On |
 State[Web3]=On | State[Web4]=On) &
(State[App1] = On | State[App2]=On ) &
(State[DB1] = On |State[DB2]=On );
```

Figure 4.10: Property for service halting

```
AG(!SplitBrain)
SplitBrain :=
(State[App1]=On & State[App2]=On) |
(State[DB1]=On & State[DB2]=On);
```

Figure 4.11: Property for data inconsistency

Monitor change operation: For an example of operations executed in cloud user side, we suppose the configuration changes in HA clusters can be executed to change monitor target in a standby (backup) virtual machine. We chose this operation for our case study because the misconfigurations in HA cluster is one of the most common causes of split-brain syndrome as described in [115]. It is also reported in [114] that the complexity in the cluster setup is the dominant factor of failover success rates.

We assume the following two faulty conditions to be undesirable situations.

(1) Service halting

If there is a fault in all of the components consisting of a layer, we regard this as a service halting failure. In our scenario, it occurs when four web servers, or two application servers or two database servers fail at the same time. The property can be represented by the formula in Figure 4.10 that was constructed based on Definition 7.

(2) Data inconsistency

When a primary component and the backup for it in a HA cluster are working at the same time, we consider that it is a situation involving data inconsistency. In our scenario, it is the case when both App1 and App2 are on, or when both DB1 and DB2 are on. Based on Definition 8, we can describe the property using the formula in Figure 4.11.

4.4.3 Evaluation of operational vulnerability by NuSMV

We now evaluate the operational vulnerability of the system in the case study using NuSMV. By implementing the system models in patterns A and B, and the properties to be verified described in the previous subsection on NuSMV, we conducted an evaluation of the operational vulnerability. For the evaluation, we used a PC with an Intel Core2 Duo E8500 CPU (3.17 GHz), 4 GB memory, Windows 7 Professional OS (32 bit), and

Table 4.1: Verification results for pattern A

	No service halting			No data inconsistency		
	Number of faults			Number of faults		
	0	1	2	0	1	2
No operation	True	**True**	False	True	True	True
Migration	True	False	False	True	True	True
Monitor change	True	False	False	True	False	False
Migration + Monitor change	True	False	False	True	False	False

Table 4.2: Verification results for pattern B

	No service halting			No data inconsistency		
	Number of faults			Number of faults		
	0	1	2	0	1	2
No operation	True	**False**	False	True	True	True
Migration	True	False	False	True	True	True
Monitor change	True	False	False	True	False	False
Migration + Monitor change	True	False	False	True	False	False

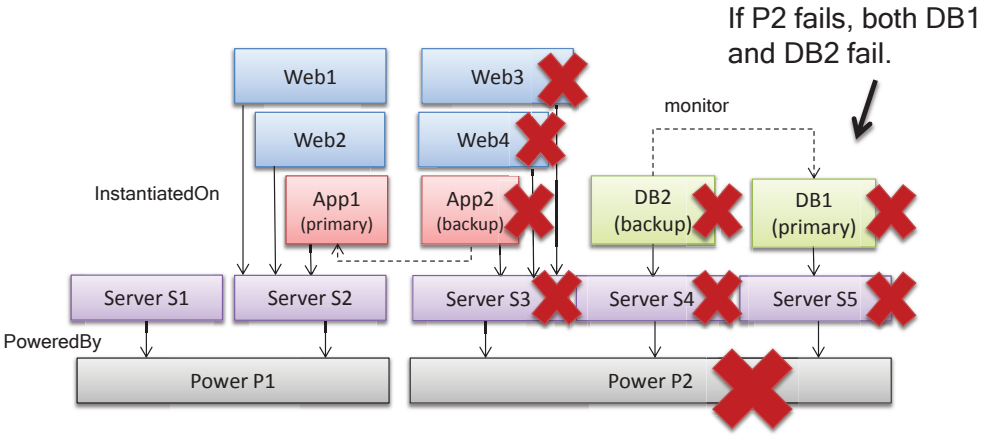
NuSMV version 2.5.4 for Windows. In the evaluation, we changed the number of possible component faults from 0 to 2. The number of operations for live migration and changing monitor target was limited to less than two. Tables 4.1 and 4.2 shows the satisfiability of the defined properties in patterns A and B, respectively. Table 4.3 shows the operational vulnerability in each pattern calculated from Tables 4.1 and 4.2.

From these results, we can summarize the evaluation results of the operational vulnerability as follows.

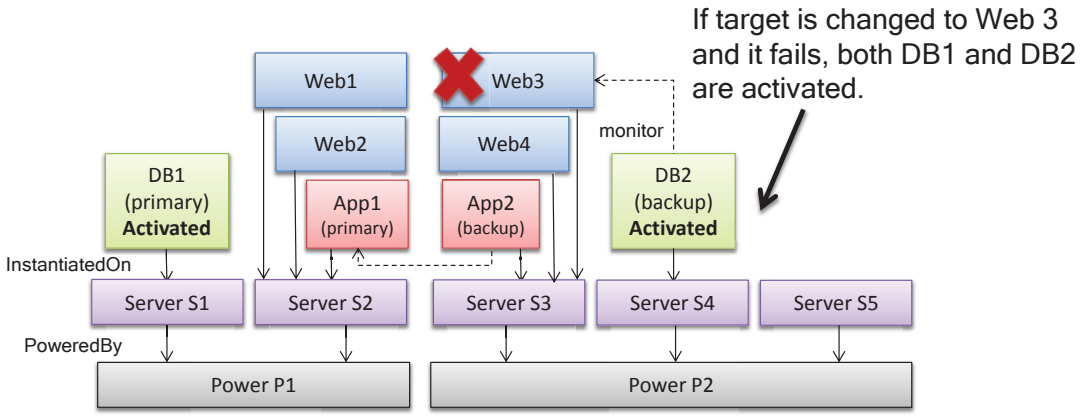
- With one component fault and no operation execution, pattern A satisfies both of the properties, while the service halting failure occurs in pattern B. This is because the physical server S4 accommodating DB2 and S5 accommodating DB1 are powered by the same power unit P2 in pattern B. If P2 fails, both DB1 and DB2 shut down, and the service cannot be available (Figure 4.12(a)). Therefore, we can conclude that there is a single point of failure in pattern B, and it is more vulnerable than

Table 4.3: Evaluation results of operational vulnerability

	Pattern A			Pattern B		
	Number of faults			Number of faults		
	0	1	2	0	1	2
No operation	**Lv.0**	**Lv.1**	Lv.2	**Lv.1**	**Lv.2**	Lv.2
Migration	Lv.1	Lv.2	Lv.2	Lv.1	Lv.2	Lv.2
Monitor change	Lv.1	Lv.3	Lv.3	Lv.1	Lv.3	Lv.3



(a) Service halting in pattern B



(b) Data inconsistency with monitor change in pattern A

Figure 4.12: Deployment patterns of virtual machines

the placement in pattern A, which does not have a single point of failure. From this result, the operational vulnerability of pattern B is level 1 (single point of failure) when there is no fault, and level 2 (service halting) if a fault happens.

- When we execute only live migration operations, it does not lead to data inconsistency (split brain). However, if we execute the operations that change the monitor target, one component fault can lead to service halting or a split brain. For example, if we change the monitoring target for the backup database server (DB2) from DB1 to Web3, then the fault of Web3 can invoke the starting up of DB2, resulting in both DB1 and DB2 working (Figure 4.12(b)). Therefore, the operation that changes the monitoring target is more risky, because it can more easily elevate the operational vulnerability to level 3 compared to live migration.

- The vulnerability levels when we executed the live migration operation and operation changing monitor target are the same in both patterns A and B. Therefore, we can conclude that the impacts of these operations are the same in these configuration

Table 4.4: Computational time (sec) for pattern A

	No service halting			No data inconsistency		
	Number of faults			Number of faults		
	0	1	2	0	1	2
No operation	1.08	1.53	6.90	1.05	1.57	6.58
Migration	1.08	4.41	10.86	1.11	4.01	97.69
Monitor change	1.10	1.73	10.84	1.06	1.71	10.20
Migration + Monitor change	1.13	8.82	66.75	1.14	8.55	118.87

Table 4.5: Computational time (sec) for pattern B

	No service halting			No data inconsistency		
	Number of faults			Number of faults		
	0	1	2	0	1	2
No operation	1.09	1.29	6.18	1.04	1.12	5.74
Migration	1.11	4.43	46.56	1.09	4.06	59.17
Monitor change	1.08	1.72	11.57	1.17	1.70	11.03
Migration + Monitor change	1.14	8.32	63.52	1.14	8.32	51.13

patterns.

Tables 4.4 and 4.5 show the computational time required for each evaluation. From these tables, we can see that we obtained the results in about one second in the best-case scenario, while it took over 100 seconds in the worst-case scenario.

4.5 Discussion

In the case study, we demonstrated that we can evaluate the operational vulnerability of a system for component faults and improper operation executions. We summarize the advantages of our approach as follows.

(1) **It enables analyses to consider the states of system components and interactions between them**

In our approach, we defined a system model using components and interdependencies between them through which the state transitions propagate to other components. By doing this, we can represent the "domino effect" of state transitions that occur in a system consisting of components with complex interdependencies. As a result, we can evaluate the operational vulnerability hidden in the system, which is difficult to identify from the static analysis of the system configurations.

(2) **It enables impact evaluation for faults and operations**

By defining the levels of operational vulnerability in information system management, we enabled the impact evaluation for the occurrence of faults or the execution of improper configuration change operations. By doing this, we can identify the

types of faults or operations that can seriously affect the system. By utilizing the information and modifying the system configuration properly, we can improve the robustness of the system for the faults.

On the other hand, to make our approach more practical, we need to solve the following problems.

(1) **Establishing efficient criteria to construct models for various operations**

While we constructed state transition models for two types of operations (live migration and changing the monitor target) on NuSMV, various types of operations can be executed in the cloud infrastructure management. Therefore, we need to clarify the general criteria or principles when modeling these operations so that we can construct models for any types of operations without ambiguity.

(2) **Establishing a method to construct smaller models**

From the evaluation of computational times in our case study, we found that the model checking is capable of determining the vulnerability in a service with a typical system configuration. However, we also found that the computational times increase significantly when we increase the number of possible fault components and operation executions. Therefore, in order to realize the evaluation for more complex and large-scale systems, we need to construct small-size models that can be verified effectively. For example, while we constructed a state transition model for each component in our case study, we may be able to reduce the number of states by grouping some components into one state transition model if the grouping does not affect the overall behavior of the system.

4.6 Summary

In order to improve the reliability in configuration changes in cloud system management, administrators have to take care of the execution of the change as well as the planning of it. However, estimating the outcome of operations for components which are interrelated dynamically is quite difficult. To solve this problem, by using state transition models and model checking approach, we proposed a method to evaluate the operational vulnerability of services on a cloud computing infrastructure for component faults and improper operation executions. In our approach, we first defined the level of vulnerability for accidental faults or improper operations based on their impact on services (service unavailable and data inconsistency). Next, we constructed a model that represents the behavior of systems by components and their interrelationships using state transition models. We enabled the evaluation of the impact of events (component faults or operations) using our model, in which state transitions can be propagated between components having interrelationships. Then, we demonstrated that we can evaluate the vulnerability of services by carrying out a case study in which the wrong target assignments for live migration and high availability cluster are considered. In this case study, we verified the system model using the model checking tool NuSMV and determined the single point of failures, possible system halting and data inconsistencies.

We are now considering the following work for future study. First, we are going to establish an efficient method for the construction of models for various operations. As discussed in the previous section, we have to model various cloud management operations such as taking snapshots of virtual machines and applying security patches to software components. Next, we aim to improve the model construction method for obtaining smaller models that can be verified with smaller computing resources (CPU time and memory size). By improving our approach regarding its complexity (e.g., the variations of configuration operations) and scalability (e.g., the size of model that can be analyzed), it will become more practical for large-scale complex cloud computing systems. We believe that by realizing these goals, our method will contribute to the improvement of the reliability of cloud system management by recognizing the risks hidden in the system structure, and by correcting the configuration beforehand to prevent service failures.

Chapter 5

Discussion

5.1 Practicality and limitation of the proposed approach

In this dissertation, we discussed how to improve the reliability in cloud system management using formal methods. In our approach, we verified the possible behavior of the target system by evaluating the system configurations and the operations executed on them using formal methods. One of the most biggest advantages of the proposed approach is that the formal methods can contribute to prevent system administrators from overlooking flaws hidden in the system configurations and the management process by making use of their capability to explore the large state space corresponding to the possible situations. On the other hand, it also has limitaions such as scalabilities. In this section, we summarize and discuss the practicality of the proposed approach for actual cloud infrastructures based on the evaluation results presented in Chapter 3 and 4.

Through the case studies in Chapter 3 and 4, we evaluated the performance of verifications and found out that our technique can be applied for a system consisting of several dozen components (physical servers or virtual machines). Therefore, we suppose that our approach can be used in the problems in the following classes, though we need to do further investigations for proving it.

- A private cloud infrastructure consisting of few racks with a few dozen physical components (servers, network switches and storages). For example, when we use Pureflex [118] system with 25-Unit rack on which we can mount 25 components, we can execute the verification by our approach for a system with one or two racks.

- A virtual system consisting of dozens of virtual machines. For example, a social game site gumi [63] handles 10 million users with 8 virtual machines for web-cum-application servers and 2 virtual machines for database servers [64]. We can also find some examples of virtual systems with several virtual machines from Amazon AWS Cloud Design Patterns [126]. We suppose large portion of small Web services can belong to this category.

On the other hand, from the viewpoint of scale and complexity, it is difficult to apply our approach to systems in the following classes.

- A cloud system having over about 50 components. For example, a private cloud system at Japan Advanced Institute of Science and Technology consists of 50 physical nodes and hundreds of virtual machines [128].

- A virtual system with hundreds nodes. For example, a system for large amount of data (e.g. hadoop [127]) can consist of hundred or thousand of virtual nodes.

We suppose that these limitations come from our straightforward modeling approach. We prepare one object corresponding to each component comprising the target system in the modeling in Chapter 3. Same thing can be said for Chapter 4 in which we use one module for each component. The increases in the number of SAT clauses are more than linear relationship with the number of components (see Table 3.1). Therefore, linear reinforcement of processing power by methods such as parallel or distributed processing for SAT solving does not seem to be an appropriate solution to scalability problem. One of the possible fundamental solutions is to simplify the model before executing analysis by formal methods. For example, some cloud infrastructures can be constructed in a uniform way (e.g. consisting of many racks of the same type on which servers of the same type are mounted). We might be able to make use of this type of repetitive structure of cloud system for constructing smaller system models by omitting or abstracting the parts having the same structure.

5.2 Possible application

While we mainly focused on the service transition stage in the system management lifecycle in this dissertation, formal methods can also be applied for the other system management stages to improve the whole management cycle. For example, it is possible that we can improve the remaining four stages (Service Strategy, Service Design, Service Operation and Continual Service Improvement) by the following approaches.

1. **Service Strategy stage**

   In this stage, it is needed to recognize the requirements for the system and specify them without ambiguity. This can be achieved by defining a goal and decompose it into several requirements. These tasks are difficult and time-consuming, since they are usually conducted manually. Some approaches to support these tasks by using formal methods have been proposed. For example, Alrajeh et al. proposed a method to elaborate requirements for a system using model checking and inductive learning [65]. In their approach, model checking is used to identify the incompleteness in a partial operational requirement specification. Then they use Inductive Logic Programming (ILP) to generate the missing part of the specification. This kind of approach can contribute to improve the efficiency in request specification.

2. **Service Design stage**

   In this stage, it is required to refine abstract service strategies and requirements to make concrete tasks. Here we have to take care not to produce contradictions between the abstract requirement and its refined task. One of the typical approaches to achieve the safe refinement is theorem proving approach. For example, Event-B and its Rodin tool can assist the refinement by providing theorem proving function

which can guarantee that the refined (lower) model satisfies the requirements in the abstract (upper) model.

3. **Service Operation stage**

   One of the most important tasks in the Service Operation stage is problem management in which system administrators have to identify the cause of problems occurred in their systems and solve them. Different from the changes planned in the Service Transition stage, the changes for problem solving should be conducted as soon as possible. Therefore, formal verificaiton approach can be used to check in a short period of time whether or not a procedure intended to be conducted to solve a problem evoke another problem by its side-effects. Some system verification methods such as [66] can be used for this purpose.

4. **Continual Service Improvement stage**

   The main task in this stage is to evaluate the performance of a system and its management to improve the service quality and cost effectiveness. From the viewpoint of service quality, it is important to keep service level objectives. Since the availability is the top of concerns for cloud users as shown in Table 2.1, evaluation and improvement of availability are quite important. Formal methods can also be used for the availability improvement. For example, Calinescu et al. proposed a technique in [58] to estimate the service availability of a cloud system based on the failure rates of components comprising the cloud system. By using their approach, cloud administrators can determine whether or not the cloud system can satisfy a given availability service level. If it is determined that the system cannot satisfy the availability rate, administrators can plan the configuration change to improve the availability.

The above approaches also suggest that formal methods can be used in various ways in the system management lifecycle. By combining our approch for Service Transition stage with the above approaches for the other stage, the effectiveness and availability of a cloud system can be improved throughout the lifecycle of system management.

5.3 Related work

As mentioned before, configuration of complex ICT systems is widely regarded as one of the key challenges in system management [16]. In this thesis, we focus on the improving the reliability of change management by avoiding unnecessary failures by two approaches: (1) configuration change planning in change design phase and (2) configuration verification in execution phase. Although quick recovery by some techniques such as recovery oriented computing [124] can be an alternative approach to improve the availability of systems, it is out of scope of this thesis, because it is not an approach to prevent failures.

5.3.1 Configuration change planning

The design choices in configuration change planning can be categorized into the following three categories: (1) autonomic management, (2) planning with procedural knowledge and (3) planning with procedure and constraints.

Autonomic management

In 2003, autonomic computing framework [125] to make systems manage themselves has been proposed. It is a technique to orchestrate the behavior of systems consisting of many autonomic components working with simple if-then type policies.

Weyns et al. [72] surveyed the researches regarding the application of formal methods for self-adaptive systems from 2000 to 2010. In this survey, it has been reported that while the most dominant application domain of formal methods in self-adaptive systems is embedded systems (46.7%), service-based systems (e.g. Web services) is in the second position (26.7%) and have gained an increasing attention since 2005. For example, Calinescu et al. [73, 74] proposed a framework to control web service resources (e.g. CPUs) dynamically based on the quantitative verification results using PRISM model checker. Xu et al. [62] proposed a method to determine the placement pattern of virtual machines on physical servers in a datacenter so that the pattern satisfies constraints such as the network bandwidth.

These researches of autonomic computing mainly focus on the realtime reaction for the situation changes (e.g. increase of user requests) without making drastic changes in system structures. Therefore, orchestrating the behavior of autonomic systems is different from our purpose which is to synthesize procedure for scheduled configuration changes consisting of several operations.

Planning with procedural knowledge

Multiple research projects have investigated the planning of procedures for system configurations using pre- and post-conditions of operations, including Plaint [17, 18] and LPG [19]. Cordeiro et al. proposed ChangeLedge framework [60], which can synthesize the configuration change procedure by refining a sequence of abstracted operation descriptions. Hagen et al. [59, 61, 109, 110] conducted thorough research regarding configuration planning in cloud systems. In [59], a fundamental framework of a hybrid approach for IT (information technology) Change Planning has been proposed. In this approach, Change Requests (CRs) are decomposed into subtasks using Hierarchical Task Network (HTN) method [111]. At the same time, the operations for objects in the target system are described by means of extended restricted state-transition systems (eSTSs), an extended version of restricted STSs [112]. Each object is in one of the three states representing its status: Installed, Started or Removed. Each atomic change request (task) is represented by a state transition with its pre- and post-conditions. The hybrid approach decomposes a given task into subtasks and synthesizes a change plan by determining the order of atomic change requests so that state changes for each objects do not conflict with each other. This work has been extended in [61] for virtualized environments. For example, it can handle preconditions for change requests such as "In order to install an application in a virtual machine (VM), it needs to be in state 'running' and the physical machine (PM) needs to be in state 'on' ". The performance and usability of proposed approach have been discussed in [109]. By comparison with other approaches such as Graphlan algorithm with planning graphs and Prodigy algorithm based on means-end alaysis, they concluded that Hierarchical Task Network outperformed the others. The HTN algorithm has been improved in [110] by improving the process in the planning such as evaluation of preconditions and determination of bindings between parameters in configuration change activities and configuration items.

These techniques only rely on procedural knowledge containing information about each operation's pre- and post-conditions. They synthesize a procedure just by connecting operations that comply with the procedural knowledge. This approach is different from ours which utilizes both of the procedural and declarative knowledge.

Planning with procedure and constraints

As mentioned above, there are some cases in which administrators have to take into account not only the procedures for making changes to configurations but also declarative constraints that should be maintained in their system. For example, some constraints in system management might be defined declaratively as "anti-pattern" or "not to do list" which are independent from procedural knowledge as shown in [113]. Actually, some researchers have started to realize the importance of using discrete and declarative constraints for system management [22], although most of their research is at an early stage and concerns the propositions of concepts or architecture.

Our approach is focusing on this category, while there are not so many approaches which are based on sound logical and mathematical foundations like ours. One of the few approaches for incorporating conditions independent of procedural knowledge into a method of planning configurations is SPiCE [20, 21] by IBM. They, however, have required programming to define these constraints and have not been able to express these constraints declaratively. We assumed that knowledge on system management could be added, removed, or modified within the lifecycles of system management due to various reasons such as changes in system management policies, emersions of new components, and the disposition of knowledge on obsolete components. Therefore, it is very disadvantageous to embed that knowledge in procedural program code, because it is too difficult to modify the embedded knowledge scattered in a planning algorithm.

5.3.2 Configuration verification

As for the research regarding configuration verification, various types of researches regarding the formal approaches for information systems have been done so far. The major research topic in this area so far is the verification of program code, not configuration management. For example, Near et al. developed a verification tool called Rubicon [79] for web applications developed by using Ruby on Rails framework [80]. Rubicon translates the Ruby codes and specifications into Alloy language so that they can be verified by using Alloy Analyzer. The authors executed the verifications for several open-source web applications and identified a previously unknown security bug in one of them. Bianculli et al. [85] used Labelled Transition System Analyser (LTSA) [86, 87] to estimate the behaviors of external service components as labelled transition systems (LTS) from the specifications of them. Artho et al. [96, 97] conducted the verification for several multi-threaded network applications such as HTTP server and WebDAV server using Java PathFinder (JPF) [99, 100]. They applied cache-based model checking approach which can backtrack the verification for the multi-threaded processes with non-deterministic behaviors. The cache captures the network traffic between the system under testing (SUT) and external processes communicating with the SUT. In the backtracking, data previously received by the SUT can be replayed by the cache when it is requested. The backtracking mechanism conbributed to reduce the state space to be explored during the verification. Their work

has been extended by Leungwattanakit et al. [98] who added a checkpointing function which enables to save and restore the states of peer nodes communicating with the target SUT.

While these program verification is quite important in improving the reliability of information systems, this is the out of the scope of this thesis regarding the system configuration management.

Some researches have been focusing on formalizing system configuration and management process in cloud computing and virtualized systems. For example, Schroeter et al. [83] proposed some models for describing requirements for multi-tenant systems. They identified the requirements for configuration models based on the fact that there are various types of stakeholders (e.g. cloud providers and tenants (users)) in multi-tenant cloud service management. Based on the analysis, they proposed three types of models: (1) an Extended Feature Model for describing the functionalities of components comprising the services, (2) a View Model defining stakeholders and their views regarding the configurations and (3) Configuration Process Model for representing the configuration steps and involved stakeholders. While the authors intended to use verification methods such as constraint satisfaction problem (CSP) solvers [84] to verify the consistency in concurrent stakeholder configurations, the actual appication still remains as future work. Antonescu et al. [88] proposed the specification language for cloud-based application. In their specifications, the structure of a cloud service is described by using objects such as assets (software, images, scripts or data necessary for the service to be executable) and regions (compute and storage resource pools at different datacenters or cloud providers) and relationships between them. Using the specification, they demonstrated the dynamic resource provisioning based on the evaluation of the service level (e.g. response time).

Different from our approach, these researches regarding the formalization of system configuration do not provide verification methods by themselves. These formalized models can be the input data for the verification methods such as the ones proposed in this thesis.

As for the system configuration verification, we can consider two approaches: (1) static configuration verification and (2) dynamic configuration verification. Related work in each category is presented as follows.

Static configuration verification

The static configuration verification focuses on determining whether or not a fixed system structure can satisfy given requirement.

One of the typical applications of formal methods for static verification is for networked distributed systems. For example, Ritchey et al. [81] analyzed the vulnerabilities in network configurations in web servers by using SMV model checker. Narain et al. [29, 30, 82] used Alloy Analyzer to identify the single point of failure in a network and correct configurations, though they concentrated on the verification for static configurations without taking into account the events which can change the status of the system.

Researches involving the application of formal verification include the work done by Bleikertz et al. [89, 90, 91, 92]. They proposed a generic way to specify and verify security goals for virtualized infrastructure. In their framework, the system to be analyzed is modeled as a graph by using AVANTSSAR Specification Language [93], and the inference rules and goals to be maintained are specified as Horn clauses. They used verification tools such as OFMC (On-the-Fly Model Checker) [94] for the verifications for three examples

of security problems: (1) Zone isolation meaning that any machines deployed in a security zone (e.g. high, base, and test secutiry zones) should not be able to communicate with a machine in the zone with different security level, (2) secure migration meaning that intruders cannot transfer a target virtual machine by migration to a host for which the intruders have administrative (root) privileges and (3) absence of single point of failure meaning that there are at least 2 network paths between hosts which should be able to communicate with each other. While they used model checking approach, the problems investigated can be regard as reachability problems in static graphs. Another verification for security issue is discussed in the work done by Almorsy et al. [95]. They represented the signature of security vulnerabilities in web applications as OCL (Object Constraint Language) constraints. By translating web application programs into abstract syntax tree (AST) representation and executing the verification for them with OCL constraints, they identified hidden flaws causing security vulnerabilities in the program codes.

The main difference between these researches and our approach is that ours can take into account the dynamic system behaviors in configuration changes. The importance of the dynamic verificaiton and related work are presented as follows.

Dynamic configuration verification

Some researches also emphasized the importance of verification in systems' behaviors at runtime [75, 76, 77, 78]. Since today's information systems face changes (e.g. changes in the system components, requirements, or development environments) which are unpredictable at the development stage, some techniques such as formal methods have been applied so that we can determine whether or not the required changes can induce potential conflicts with the systems' specifications. In fact, as shown in [57], a large number of management operations (live migration and taking snapshot) are executed in cloud computing data centers every day. Therefore, the importance of dynamic verification will increase along with the prevalation of cloud computing.

With respect to the analysis approach that applies to formal methods, Calinescu et al. [58] proposed a modeling method for service availability, and analyzed the model using a PRISM probabilistic model checker, while they do not consider the impact of operations that can change the configuration of systems.

Etchevers [101] and Salaun et al. [102, 103, 129] have also conducted the verification for distributed cloud applications. They proposed a framework named VAMP (Virtual Application Management Platform) which can deploy services consisting of several virtual machines deployed on some physical servers. They implemented a deployment process and a self-configuration protocol used for communications between virtual machines so that they can bind and compose the services in the autonomic way. Since the interactions between autonomic virtual machines can be complex, they executed the verifications for properties such as "all mandatory client interfaces are connected to servers". In the verification, first they generate a LTS from the specification and the target application by using CADP (Construction and Analysis of Distributed Processes) exploration toolbox [104]. Then they conducted the verificaition with EVALUATOR model checker [105] in the toolbox. This research is rather the verification of autonomic protocols than the verification for scheduled system management process we focus on.

As for the verification of change plans, in [107] inconsistencies between planned change requests and system configurations have been discussed. They supposed that time-lags

Table 5.1: Comparison with Bleikertz's and Hagen's approaches

Work	Advantages	Disadvantages
Bleikertz	Applied to various types of scenarios (e.g. secure migration and single point of failure).	Only for static verification for the constraints which do not take into account the dynamic intaractions between components.
Hagen	High scalability for planning and verification for configuration changes.	Handle only procedural knowledge (cannot use declarative constraints).
Proposed	(1) Able to take into account both of procedural knowledge and declarative constraints. (2) Able to analyze domino-effects triggered by operations or failures.	Difficult to apply to large scale systems.

between the change planning and the execution of the planned change might make the change plan obsolate and inconsistent with updated system configurations. They discussed three types of change plan adaptions: (1) Replanning to generate a new plan for scratch, (2) Plan adaption to change some parts of the plan and (3) Parameter adaption to assign new values to parameters to make the plan executable (e.g. "If a physical machine originally chosen does not have enough memory to accommodate a virtual machine any more, alternative physical machine has to be chosen"), while only the solution for the parameter adaption has been provided. In [108], the conflicts between change requests have been discussed. The problems to be solved here is to identify change requests which can make other change requests infeasible (e.g. a change request for changing the IP address for a server can make another change request involving the access to the server infeasible). The application example of the plan verification for Amazon EC2 outage has been demonstrated in [59]. While the applied algorithm was tailor-made for the specific problem, it has been enhanced and implemented in [71] as a specific purpose model checker utilizing partial-order reduction method. Through a case study of Amazon outage, they evaluated the performance and claimed that their implementation outperformed general-purpose model checkers such as SPIN and NuSMV. While this approach seems quite similar to ours, it does not take into account the dynamic interactions between components comprising a system. In our vulnerability evaluation approach, the interaction between components (e.g. a failure of a component can invoke the start up of its backup) is an important factor.

5.3.3 Advantages and disadvantages

Here we summarize the advantages and disadvantages of our approach by comparing it with related work described above. For comparison, we chose two approaches done by Bleikertz et al. [89, 90, 91, 92] and Hagen et al. [59, 61, 71, 106, 107, 108, 109, 110], because their targets (information systems consisting of nodes playing various functions)

and approaches (model checking) are quite similar to ours. The comparison is summarized in Table 5.1. First, in Bleikertz's work, they demonstrated various scenarios in which verifications were conducted by using various types of model checkers. They also evaluated the performance (time measurement) for analysis for the zone isolation scenario. One of the notable weakness of their approach is that all scenarios are rather static than dynamic. Therefore, it cannot handle scenarios involving dynamic configuration changes or status changes in components of target systems. Next, in Hagen's work, scalability in configuration planning and verification has been thoroughly investigated. Especially, in [71] they implemented their own model checker utilizing partial-order reduction approach and compared the performance with general-purpose model checkers such as SPIN and NuSMV in real Amazon outage scenario presented in [59]. By concentrating on the procedural operation knowledge with pre- and post-conditions, they achieved high scalability. Conversely, their approach cannot use declarative constraints in their planning and verification along with procedural knowledge. Comparing with these two approaches, our approach can take into account more vairous factors than only procedural operation knowledge. For example, our configuration procedure synthesis method can construct a configuration plan which can avoid declarative constraints. This is one of the advantages of ours because sometimes constraints in system management are defined declaratively as "anti-pattern" or "not to do list" which are independent from procedural knowledge as shown in [113]. In addition, our configuration verification method takes into account domino-effect which is a sequence of state changes triggered by some events such as component failures. This type of behavior analysis is difficult to do by using static configuration verification. On the other hand, one of the biggest drawbacks of our approach for practicability is scalability. We evaluated our approaches in small case study scenarios. While we improved the procedure synthesis algorithm by using intermediate goal approach, we need to improve its scalability so that we can make our approach more practical for large-scale system analysis.

Chapter 6

Conclusion and future work

6.1 Summary

Along with the growth of cloud computing services, the role of cloud computing in our society has been becoming more and more important. Therefore, keeping the availability of cloud services are utmost importance in cloud system management. However, cloud service outages caused by human errors such as misconfigurations and executing improper operations happen every day. Therefore, it is indispensable to prevent service failures by assisting system administrators using technical approaches.

Based on this background, in this dissertation we demonstrated how to improve the reliability in cloud system management through case studies for the following two types of approaches based on formal methods.

1. **Configuration procedure synthesis**

   We proposed a framework for synthesizing configuration change procedures. In this framework, first we prepare three types of knowledge related to system management: (1) executable operations with their pre- and post-conditions, (2) declarative constraints to be satisfied during the configuration change procedures and (3) goal conditions to be achieved by executing the synthesized procedure. Next these types of knowledge are translated into Alloy language. Then we input the knowledge into Alloy Analyzer model finder which can identify models (or instances) to satisfy given conditions. In other words, Alloy Analyzer identifies the sequence of operations which can achieve goal conditions without violating the declarative constraints. This sequence of operations is output as the synthesized configuration change procedure. In order to reduce the memory size to synthesize the procedures, we devide the path from the initial state to the goal state satisfying given goal conditions into shorter paths with defining intermediate states. By shortening the paths of state changes to be synthesized, we reduced size of memories required to procedure synthesis.

2. **Operational vulnerability evaluation**

   We proposed a method to evaluate the operational vulnerability of services on a cloud computing infrastructure for component faults and improper operation executions. In our approach, first we defined the level of operational vulnerability of services based on the seriousness of situations. The operational vulnerability scale

has four levels: (Level 0) safe state, (Level 1) state with single point of failure, (Level 2) service unavailable state and (Level 3) data inconsistency state. Next we constructed system models representing the state transitions which can be triggered by component failures or operation executions in a cloud infrastructure. In this model, we take into account the interrelationships between components comprising the cloud infrastructure. By executing the verification of this model using NuSVM model checker, we analyzed the configuration of the cloud infrastructure and determined the impact of component failures or operation executions on the cloud infrastructure. From the verification results, we can determine the vunlerability level of the infrastructure and identify the weakness hidden in the configuration of the infrastructure.

These two approaches complement with each other. While the configuration procedure synthesis approach can prevent some misconfigurations by synthesizing the procedure satisfying given conditions, it might not be able to prevent all types of misconfigurations from some reasons such as the flaws in the definition of declarative constraints. However, even if some misconfigurations occur, operational vulnerability evaluation approach will be able to identify some of them. By combining different approaches, we can expand the coverages of misconfiguration preventions and improve the reliability of system management processes.

The main contribution of this dissertation is that we demonstrated how to apply formal methods for cloud management. In detail, we showed how to construct and analyze the system models representing the system configurations and the changes triggered by operation executions or components faults using the framework of formal methods. By using the system models, we demonstrated that we can construct the operation procedures and determine the operational vulnerability hidden in the system configurations.

6.2 Future work

We are now considering the following work for the future.

- **Assisting translation of system management knowledge into formal representations**

  In the proposed approaches, we suppose that system configuration information can be automatically derived from configuration management database. Therefore, we can construct the function to translate system configuration information into formal descriptions such as Alloy language or NuSMV model description. However, we defined the possible state transitions triggered by failures or operation executions manually. For administrators who are not familiar with logical representations, it might be difficult to define pre- and post-conditions of executable operations precisely. Therefore, some technologies which can assist or automate the transition of experts' knowledge regarding executable operations into formal representations can lessen the difficulties in the modeling.

- **Improving the scalability of analysis and verification**

  From the evaluation results, we conclude that our approaches still have limitations in the scalability. In the operation procedure synthesis, we reduced the memory size

required for the synthesis by dividing the sequence of operations into shorter paths. However, we still need to improve computational time and memory size requred for the analysis if we are going to apply our approach directly for mega-scale cloud computing infrastructure. There can be several approaches to solve this problem. For example, we might be able to reduce the size of system model by abstracting some components comprising the cloud system if they are not related to the events (faults or operations) directly. We also might be able to limit state transitions by limiting the possible events or the order of them to reduce state space to be explored.

We can easily imagine that there is no technology which can solve all problems in system management tasks. By combining several complemental techniques such as the procedure synthesis and the operational vulnerability evaluation proposed in the dissertation, the reliability in system operations and management can be improved efficiently.

Bibliography

[1] Rich Miller, "Report: Google Uses About 900,000 Servers", http://www.datacenterknowledge.com/archives/2011/08/01/report-google-uses-about-900000-servers/ , 2011.

[2] Huan Liu, "Amazon data center size", http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/ , 2012.

[3] Google, "Search Engine Strategies Conference, Conversation with Eric Schmidt hosted by Danny Sullivan", http://www.google.com/press/podium/ses2006.html , 2006.

[4] Amazon Web Services, http://aws.amazon.com

[5] Google App Engine, https://developers.google.com/appengine

[6] Cloud computing (wikipedia), http://en.wikipedia.org/wiki/Cloud_computing

[7] Peter Mell and Timothy Grance, The NIST Definition of Cloud Computing, 2011.

[8] itSMF, An Introductory Overview of ITIL V3, 2007.

[9] Amazon EC2 SLA, http://aws.amazon.com/ec2-sla

[10] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram and Shankar Pasupathy, "An empirical study on configuration errors in commercial and open source systems", Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11), 2011.

[11] Taro Kurita, Miki Chiba, and Yasumasa Nakatsugawa, "Application of a Formal Specification Language in the Development of the "Mobile FeliCa" IC Chip Firmware for Embedding in Mobile Phone", Lecture Notes in Computer Science, Volume 5014, pp 425-429, 2008.

[12] Jeannette M. Wing, "A Specifier's Introduction to Formal Methods", Computer, Vol.23, No.9, pp.8-22, 1990.

[13] Event-B and the Rodin Platform, http://www.event-b.org/index.html

[14] Symantec Corporation: State of the Data Center Report 2007, http://www.symantec.com/ , 2007.

[15] D. Jackson, Software Abstractions: Logic, Language, and Analysis, The MIT Press, 2006.

[16] Aaron B. Brown, Alexander Keller, Joseph L. Hellerstein, "A model of configuration complexity and its applications to a change management system", Proceedings of 9th IFIP/IEEE International Symposium on Integrated Network Management (IM2005), 2005.

[17] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf, "Deployment and dynamic reconfiguration planning for distributed software systems", Software Quality Control, Vol.15, No.3, pp.265-281, 2007.

[18] Naveed Arshad, "Automated Dynamic Reconfiguration using AI Planning", Proceedings of 19th IEEE International Conference on Automated Software Engineering (ASE 2004), 2004.

[19] Alfonso Gerevini and Ivan Serina, "LPG: a Planner based on Planning Graphs with Action Costs", Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS'02), AAAI Press, 2002.

[20] Tamar Eilam, Michael Kalantar, Alexander Konstantinou and Giovanni Pacifici, "Model-Based Automation of Service Deployment in a Constrained Environment", IBM Research Report, 2004.

[21] Kaoutar El Maghraoui, Alok Meghranjani, Tamar Eilam, Michael Kalantar, and Alexander V. Konstantinou, "Model driven provisioning: bridging the gap between declarative object models and procedural provisioning tools", Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware (Middleware '06), pp.404-423, Springer-Verlag, 2006.

[22] The Rise and Rise of the Declarative Datacentre, Microsoft Research Technical Report, MSR-TR-2008-61, 2008.

[23] DMTF CMDB Federation Working Group, http://www.dmtf.org/

[24] itSMF - The IT Service Management Forum, http://www.itsmf.co.uk/

[25] Akira Katsuno, Satoshi Tsuchiya and Motomitsu Adachi, "TRIOLE Organic Computing Architecture", Fujitsu Scientific and Technical Journal, Vol.43, No.4, pp.412-419, 2007.

[26] Alloy Analyzer 4, available at http://alloy.mit.edu/alloy4/

[27] SAT4J, available at http://www.sat4j.org/

[28] The Minisat Page, http://minisat.se/

[29] Sanjai Narain, "Network configuration management via model finding", Proceedings of the 19th conference on Large Installation System Administration Conference (LISA '05), Vol.19, USENIX Association, 2005.

[30] Ian Warren, Jung Sun, Sanjev Krishnamohan and Thiranjith Weerasinghe, "An Automated Formal Approach to Managing Dynamic Reconfiguration", Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE2006), pp.37-46, 2006.

[31] Xen, available at http://www.xen.org/

[32] Interstage Application Server, http://www.fujitsu.com/global/services/software/interstage/apserver/

[33] Amazon Elastic Compute Cloud (Amazon EC2), http://aws.amazon.com/ec2/

[34] Apache AXIS2, available at http://ws.apache.org/axis2/

[35] Kazuhiro Nakamura, Tomohiro Naruse, Kazuyoshi Takagi and Naofumi Takagi, "Efficient Translation of Logic Circuits to CNF Formulae with DBB", IPSJ SIG Technical Reports, 2007-SLDM-129, 2007.

[36] R. Reiter, "The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression," Artificial Intelligence and the Mathematical Theory of Computation, Academic Press, 1991.

[37] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", IEEE Transactions on Systems Science and Cybernetics SSC4 (2), pp. 100-107, 1968.

[38] G. B. Dantzig, Linear Programming and Extensions, Princeton University Press, Princeton, 1963.

[39] Jing Sun, Hongyu Zhang, Hai Wang, "Formal Semantics and Verification for Feature Modeling", Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005), pp.303-312, 2005.

[40] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," Technical Report No. UCB/EECS-2009-28, University of California at Berkley, USA, Feb. 10, 2009.

[41] FirstServer, overview and causes of the large scale failure (midterm report), 2012, http://support2.fsv.jp/urgent/ report.html (in Japanese)

[42] Data on 5,700 firms lost by Yahoo unit,
http://www.japantimes.co.jp/text/nb20120627a5.html, Japan Times, 2012.

[43] Amazon, Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region, 2011, http://aws.amazon.com/message/65648/

[44] Xen, http://xen.org/

[45] VMware, http://www.vmware.com/

[46] Google Compute Engine, http://cloud.google.com/products/compute-engine.html

[47] The reason why Fujitsu's cloud is chosen: Realization of 3-tier systems on Cloud, http://jp.fujitsu.com/solutions/cloud/iaas/fgcps5/reasons/01.html (in Japanese)

[48] Naotaka Owada, Why Systems Go Down, Nikkei BP press, 2009. (in Japanese)

[49] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield, "Live migration of virtual machines," Proceedings of 2nd ACM/USENIX Symposium on Network Systems Design and Implementation, pp. 273-286, 2005.

[50] Edmund M. Clarke, Orna Grumberg and Doron Peled, Model Checking, The MIT Press, 1999.

[51] Bryant, R.E., "Graph-based algorithms for Boolean function manipulation," IEEE Transactions of Computer, Vol.C-35, No.8, pp.677-691 (1986).

[52] G. J. Holzmann, "The model checker SPIN," IEEE Transactions on Software Engineering, Vol.23, No.5, 1997.

[53] Marta Kwiatkowska, Gethin Norman, and David Parker, "Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach," International Journal on Software Tools for Technology Transfer (STTT), 6(2), pp. 128-142, 2004.

[54] NuSMV, http://nusmv.irst.itc.it/

[55] Elastic Load Balancing, http://aws.amazon.com/elasticloadbalancing/

[56] Amazon Simple Storage Service (Amazon S3), http://aws.amazon.com/s3/

[57] Vijayaraghavan Soundararajan and Jennifer M. Anderson, "The impact of management operations on the virtualized datacenter", Proceedings of the 37th annual international symposium on Computer architecture (ICSA'10), 2010.

[58] Radu Calinesu, Shinji Kikuchi and Kenneth Johnson, "Compositional Reverification of Probabilistic Safety Properties for Large-Scale Complex IT Systems," Development, Operation and Management of Large-Scale Complex IT Systems, volume 7539 of LNCS, Springer, 2012.

[59] Sebastian Hagen, Michael Seibold, and Alfons Kemper, "Efficient verification of IT change operations or: How we could have prevented Amazon's cloud outage," Proceedings of Network Operations and Management Symposium (NOMS), pp.368-376, 2012.

[60] Weverton Luis da Costa Cordeiro, Guilherme Sperb Machado, Fabricio Girardi Andreis, Alan Diego dos Santos, Cristiano Bonato Both, Luciano Paschoal Gaspary, Lisandro Zambenedetti Granville, Claudio Bartolini, David Trastour, "ChangeLedge: Change design and planning in networked systems based on reuse of knowledge and automation," Computer Networks, Volume 53, Issue 16, pp. 2782-2799, 2009.

[61] Sebastian Hagen, Alfons Kemper, "Model-Based Planning for State-Related Changes to Infrastructure and Software as a Service Instances in Large Data Centers," Proceedings of 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD), pp.11-18, 2010.

[62] Jielong Xu, Jian Tang, Kevin Kwiat, Weiyi Zhang and Guoliang Xue, "Survivable Virtual Infrastructure Mapping in Virtualized Data Centers," Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD 2012), pp.196-203, 2012.

[63] gumi, http://gu3.co.jp/

[64] AWS Case Study: gumi Inc., http://aws.amazon.com/jp/solutions/case-studies/gumi-english/

[65] Dalal Alrajeh, Jeff Kramer, Alessandra Russo and Sebastian Uchitel, "Elaborating Requirements using Model Checking and Inductive Learning", IEEE Transactions on Software Engineering, vol.39, no.3, pp.361-383, 2013.

[66] Shinji Kikuchi, Satoshi Tsuchiya, Atuji Sekiguchi and Tsuneo Katsuyama, "Formal Description and Verification of System Management Processes Using UML", IPSJ Journal, No.50, Vol.2, pp.637-650, 2009. (in Japanese)

[67] Kevin Lano, Specification in B: An Introduction using the B Toolkit, World Scientific Publishing Company, Imperial College Press, 1996.

[68] Paris Metro Line 14 (wikipedia), https://en.wikipedia.org/wiki/Paris_Metro_Line_14

[69] M. Ben-Ari, Z.Manna, and A.Pnueli, "The temporal logic of branching time", Acta Informatica 20, pp.207-226, 1983.

[70] Fumio Machida, Masahiro Kawato, and Yoshiharu Maeno, "Just-in-Time Server Provisioning Using Virtual Machine Standby and Request Prediction", Proceedings of the 2008 International Conference on Autonomic Computing (ICAC '08), IEEE Computer Society, 2008.

[71] Sebastian Hagen, Algorithms for the Efficient Verification and Planning of Information Technology Change Operations, Ph.D. Dissertation, Technische Universitat Munchen, 2013.

[72] Danny Weyns, M. Usman Iftikhar, Didac Gil de la Iglesia, and Tanvir Ahmad, "A survey of formal methods in self-adaptive systems", Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering (C3S2E '12), pp.78-79, ACM, 2012.

[73] Radu Calinescu and Marta Kwiatkowska, "Using quantitative analysis to implement autonomic IT systems", Proceedings of the 31st International Conference on Software Engineering (ICSE '09), pp.100-110, 2009.

[74] Radu Calinescu, "Reconfigurable service-oriented architecture for autonomic computing", International Journal on Advances in Intelligent Systems, Vol.2 pp.38-57, 2009.

[75] Radu Calinescu and Shinji Kikuchi, "Formal methods @ runtime", Proceedings of the 16th Monterey conference on Foundations of computer software: modeling, development, and verification of adaptive systems (FOCS'10), pp.122-135, 2010.

[76] Gordon Blair, Nelly Bencomo, Robert B. France, "Models @ run.time", Computer, vol.42, no.10, pp.22-27, 2009.

[77] Antonio Filieri, "QoS verification and model tuning @ runtime", Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11), pp.408-411, 2011.

[78] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli, "Run-time efficient probabilistic model checking", Proceedings of the 33rd International Conference on Software Engineering (ICSE '11), pp.341-350, 2011.

[79] Joseph P. Near and Daniel Jackson, "Rubicon: bounded verification of web applications", Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12), 2012.

[80] Ruby on Rails, http://rubyonrails.org/

[81] Ronald W. Ritchey and Paul Ammann, "Using Model Checking to Analyze Network Vulnerabilities", Proceedings of the 2000 IEEE Symposium on Security and Privacy (SP '00). IEEE Computer Society, 2000.

[82] Sanjai Narain, Y.H. Alice Cheng, Alex Poylisher and Rajesh Talpade, "Network Single Point of Failure Analysis via Model Finding", Proceedings of First Alloy Workshop, 2006.

[83] Julia Schroeter, Peter Mucha, Marcel Muth, Kay Jugel, and Malte Lochau, "Dynamic configuration management of cloud-based applications", Proceedings of the 16th International Software Product Line Conference (SPLC '12), Vol.2, pp.171-178, 2012.

[84] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortes, "Using java CSP solvers in the automated analyses of feature models", Proceedings of the 2005 international conference on Generative and Transformational Techniques in Software Engineering (GTTSE'05), pp.399-408, Springer-Verlag, 2005.

[85] Domenico Bianculli, Dimitra Giannakopoulou, and Corina S. Pasareanu, "Interface decomposition for service compositions", Proceedings of the 33rd International Conference on Software Engineering (ICSE '11), pp.501-510, ACM, 2011.

[86] Jeff Magee and Jeff Kramer, Concurrency: State Models and Java Programs, John Wiley & Sons, 2006.

[87] LTSA - Labelled Transition System Analyser, http://www.doc.ic.ac.uk/ltsa/

[88] Alexandru-Florian Antonescu, Philip Robinson, and Torsten Braun, "Dynamic Topology Orchestration for Distributed Cloud-Based Applications", Proceedings of the 2012 Second Symposium on Network Cloud Computing and Applications (NCCA '12), pp.116-123, IEEE Computer Society, 2012.

[89] Soren Bleikertz, Matthias Schunter, Christian W. Probst, Dimitrios Pendarakis, and Konrad Eriksson, "Security audits of multi-tier virtual infrastructures in public infrastructure clouds", Proceedings of the 2010 ACM workshop on Cloud computing security workshop (CCSW '10), pp.93-102, ACM, 2010.

[90] Soren Bleikertz, Thomas Grob, and Sebastian Modersheim, "Automated verification of virtualized infrastructures", Proceedings of the 3rd ACM workshop on Cloud computing security workshop (CCSW '11), ACM, pp.47-58, 2011.

[91] Soren Bleikertz and Thomas Grob, "A Virtualization Assurance Language for Isolation and Deployment", Proceedings of the 2011 IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY '11), pp.33-40, IEEE Computer Society, 2011.

[92] Soren Bleikertz, Thomas Grob, Matthias Schunter, and Konrad Eriksson, "Automated information flow analysis of virtualized infrastructures", Proceedings of the 16th European conference on Research in computer security (ESORICS'11), pp.392-415, Springer-Verlag, 2011.

[93] AVANTSSAR: ASLan final version with dynamic service and policy composition. Deliverable D2.3, Automated Validation of Trust and Security of Service-oriented Architectures (AVANTSSAR), 2010. http://www.avantssar.eu/pdf/deliverables/avantssar-d2-3.pdf

[94] David Basin, Sebastian Modersheim, and Luca Vigano, "OFMC: A symbolic model checker for security protocols", International Journal of Information Security, Vol.4 No.3, pp.181-208, 2005.

[95] Mohamed Almorsy, John Grundy, and Amani S. Ibrahim, "Supporting automated vulnerability analysis using formalized vulnerability signatures", Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), pp.100-109, ACM, 2012.

[96] Cyrille Artho, Watcharin Leungwattanakit, Masami Hagiya, Yoshinori Tanabe, "Efficient Model Checking of Networked Applications", TOOLS-EUROPE 2008, Lecture Notes in Business Information Processing, Vol.11, pp.22-40, 2008.

[97] Cyrille Artho, Watcharin Leungwattanakit, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto, "Cache-Based Model Checking of Networked Applications: From Linear to Branching Time", Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09), pp.447-458, IEEE Computer Society, 2009.

[98] Watcharin Leungwattanakit, Cyrille Artho, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto, "Model Checking Distributed Systems by Combining Caching and Process Checkpointing", Proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp.103-112, 2011.

[99] NASA, Java PathFinder, http://javapathfinder.sourceforge.net/

[100] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda, "Model Checking Programs", Automated Software Engineering, Vol.10, No.2, pp.203-232, 2003.

[101] Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, and Noel de Palma, "Self-Configuration of Distributed Applications in the Cloud", Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing (CLOUD '11). pp.668-675, IEEE Computer Society, 2011.

[102] Gwen Salaun, Xavier Etchevers, Noel De Palma, Fabienne Boyer, and Thierry Coupaye, "Verification of a self-configuration protocol for distributed applications in the cloud", Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12), pp.1278-1283, ACM, 2012.

[103] Gwen Salaun, Fabienne Boyer, Thierry Coupaye, Noel De Palma, Xavier Etchevers, and Olivier Gruber, "An experience report on the verification of autonomic protocols in the cloud", Innovations in Systems and Software Engineering, Vol.9, No.2, pp.105-117, Springer-Verlag, 2013.

[104] Hubert Garavel, Frederic Lang, Radu Mateescu, and Wendelin Serwe, "CADP 2010: a toolbox for the construction and analysis of distributed processes", Proceedings of the 17th international conference on Tools and algorithms for the construction and analysis of systems: part of the joint European conferences on theory and practice of software (TACAS'11/ETAPS'11), pp.372-387, Springer-Verlag, 2011.

[105] EVALUATOR4, http://cadp.inria.fr/man/evaluator4.html

[106] Sebastian Hagen, Nigel Edwards, Lawrence Wilcock, Johannes Kirschnick, and Jerry Rolia, "One Is Not Enough: A Hybrid Approach for IT Change Planning", Proceedings of the 20th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM '09), pp.56-70, Springer-Verlag, 2009.

[107] Sebastian Hagen and Alfons Kemper, "Facing the Unpredictable: Automated Adaption of IT Change Plans for Unpredictable Management Domains", Proceedings of 6th IEEE/IFIP International Conference on Network and Service Management (CNSM 2010), 2010.

[108] Sebastian Hagen and Alfons Kemper, "Towards Solid IT Change Management: Automated Detection of Conflicting IT Changes", Proceedings of 12th IEEE/IFIP International Symposium on Integrated Network Management (IM 2011), 2011.

[109] Sebastian Hagen and Alfons Kemper, "A performance and usability comparison of automated planners for IT change planning", Proceedings of the 7th International Conference on Network and Services Management (CNSM '11), pp.143-151, 2011.

[110] Sebastian Hagen, Weverton Luis da Costa Cordeiro, Luciano Paschoal Gaspary, Lisandro Zambenedetti Granville, Michael Seibold, and Alfons Kemper, "Plannig in the Large: Efficient Generation of IT Change Plans on Large Infrastructures", Proceedings of 8th International Conference on Network and Service Management (CNSM 2012), 2012.

[111] Kutluhan Erol, James Hendler and Dana S. Nau, "HTN Planning: Complexity and Expressivity", Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), vol.2, pp.1123-1128, AAAI Press/MIT Press, 1994.

[112] Dana Nau, Malik Ghallab, and Paolo Traverso, Automated Planning: Theory & Practice, Morgan Kaufmann Publishers Inc., 2004.

[113] Dana Glasner and Vugranam C. Sreedhar, "Configuration Reasoning and Ontology For Web", Proceedings of IEEE International Conference on Services Computing (SCC 2007), pp.387-394, 2007.

[114] Klaus Schmidt, High Availability and Disaster Recovery, Springer, 2006.

[115] VMware, Resolving two active servers, http://kb.vmware.com/kb/1014405 , 2011.

[116] IDC Japan, Forecast of domestic private cloud market (in Japanese), http://www.idcjapan.co.jp/Press/Current/20130812Apr.html , 2013.

[117] MM Research Institute, Trend of demand for domestic cloud service (in Japanese) , http://www.m2ri.jp/newsreleases/main.php?id=010120130828500 , 2013.

[118] IBM, IBM PureFlex System, http://www-03.ibm.com/systems/pureflex/express/index.html , 2012.

[119] IBM, IBM PureFlex System configurations, http://public.dhe.ibm.com/common/ssi/ecm/en/wad12346usen/WAD12346USEN.PDF , 2012.

[120] Emerson, Knurr CoolLoop, http://www.emersonnetworkpower.com/en-EMEA/Products/RACKSANDINTEGRATEDCABINETS/RackCooling/Pages/KnurrCoolLoop10to30kWCoolingPower.aspx , 2013.

[121] Jing Xu and Jose A. B. Fortes, "Multi-objective Virtual Machine Placement in Virtualized Data Center Environments", Proceedings of 2010 IEEE/ACM International Conference on Green Computing and Communications & 2010 IEEE/ACM International Conference on Cyber, Physical and Social Computing (GREENCOM-CPSCOM '10), pp.179-188, 2010.

[122] Ian Whalley and Malgorzata Steinder, "Licence-aware management of virtual machines", Proceedings of 12th IFIP/IEEE International Symposium on Integrated Network Management (IM2011), pp.169-176, 2011.

[123] Hewlett-Packard, HP Storage Essentials, http://h18006.www1.hp.com/storage/software/srmgt/integrations/4AA2-0348ENW.PDF , 2008.

[124] George Candea, Aaron B. Brown, Armando Fox, and David Patterson, "Recovery-Oriented Computing: Building Multitier Dependability", Computer, Vol.37, No.11, pp.60-67, 2004.

[125] Jeffrey O. Kephart and David M. Chess, "The Vision of Autonomic Computing", Computer, Vol.36, No.1, pp.41-50, 2003.

[126] Amazon, AWS Cloud Design Patterns, http://en.clouddesignpattern.org/index.php/Main_Page , 2012.

[127] Apache Hadoop, http://hadoop.apache.org/ , 2013.

[128] Fujitsu, Case Study JAIST, http://www.fujitsu.com/downloads/GBG/casestudies/CS-JAIST-en.pdf , 2011.

[129] Rim Abid, Gwen Salaun, Francesco Bongiovanni, and Noel De Palma, "Verification of a Dynamic Management Protocol for Cloud Applications", Proceedings of 11th International Symposium of Automated Technology for Verification and Analysis (ATVA 2013), pp.178-192, 2013.

Publications

[1] Shinji Kikuchi and Satoshi Tsuchiya: "Configuration Procedure Synthesis for Complex Systems Using Model Finder", Proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2010), March 2010.

[2] Shinji Kikuchi and Toshiaki Aoki, "Evaluation of Operational Vulnerability in Cloud Service Management using Model Checking", Proceedings of the 7th International Symposium on Service Oriented System Engineering (SOSE 2013), March 2013.

[3] Shinji Kikuchi Satoshi Tsuchiya, and Kunihiko Hiraishi, "Configuration Change Procedure Synthesis Using Model Finder", IEICE Transactions on Information and Systems, Vol.E96-D, No.8, pp.1696-1706, 2013.