Synthesis of Configuration Change Procedure Using Model Finder*

Shinji KIKUCHI†‡†, Satoshi TSUCHIYA†, Nonmembers, and Kunihiko HIRAIHISHI††, Member

SUMMARY Managing the configurations of complex systems consisting of various components requires the combined efforts by multiple domain experts. These experts have extensive knowledge about different components in the system they need to manage but little understanding of the issues outside their individual areas of expertise. As a result, the configuration constraints, changes, and procedures specified by those involved in the management of a complex system are often interrelated with one another without being noticed, and their integration into a coherent procedure for configuration represents a major challenge. The method of synthesizing the configuration procedure introduced in this paper addresses this challenge using a combination of formal specification and model finding techniques. We express the knowledge on system management with this method, which is provided by domain experts as first-order logic formulas in the Alloy specification language, and combine it with system-configuration information and the resulting specification. We then employ the Alloy Analyzer to find a system model that satisfies all the formulas in this specification. The model obtained corresponds to a procedure for system configurations that satisfies all expert-specified constraints. In order to reduce the resources needed in the procedure synthesis, we reduce the length of procedures to be synthesized by defining and using intermediate goal states to divide operation procedures into shorter steps. Finally, we evaluate our method through a case study on a procedure to consolidate virtual machines.

key words: system configuration, procedure synthesis, planning, model finder, alloy analyzer

1. Introduction

Large-scale complex IT systems such as data centers need to be available 24 hours a day, seven days a week, 365 days a year (24/7/365) for users, despite incessant changes in their requirements, continual variations in user demands and management policies, and frequent installations of new components. Ensuring that these systems operate effectively at all times necessitates regular updates in their configurations. This represents a major challenge for the administrators of today’s large-scale complex IT systems, as configuration changes require a coherent combination of area-specific configuration procedures produced by multiple domain experts. For data centers for instance, such procedures may be proposed by network, server, OS, and database 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. Lack of consultation may lead to other experts ignoring this constraint in their planning, and ultimately to communication failures in the re-configured IT 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 discrete constraints to avoid violating them. 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 [1]. 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 paper, 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 [2]. Next, we derive information on current system configurations from a CMDB 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 determines the true 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 has to synthesize the procedures for smaller configuration steps instead of synthesizing the whole procedure steps.

This paper is an experience paper showing how to apply formal methods for IT system management. In detail, the main contribution of this paper is to demonstrate procedure synthesis for configuration change operations using Alloy Analyzer through a case study scenario.

The rest of this paper is organized as follows. First, Sect. 2 surveys related work, and the Alloy Analyzer model finder we used for our method of synthesis is introduced in Sect. 3. Next, Sect. 4 presents our method along with how knowledge on system management is represented. Then, we explain how it works through a case study in Sect. 5.
Sect. 6 explains our evaluation, we give some discussions in Sect. 7. Finally, Sect. 8 concludes the paper and outlines future challenges.

2. Related Work

The configuration of complex IT systems is widely regarded as one of the key challenges in system management [3]. As such, multiple research projects have investigated the planning of procedures for system configurations, including Plaint [4], [5] and LPG [6]. Most of 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. However, various experts managing their systems have different areas of expertise and knowledge on not only the procedures for making changes to configurations but also declarative constraints that should be maintained in their system. Therefore, we believe that it is insufficient to only use procedural knowledge to design configuration procedures for today’s IT systems, and that knowledge describing declarative conditions independent of procedural knowledge is urgently required.

To incorporate conditions independent of procedural knowledge into a method of planning configurations, several approaches, such as SPiCE [7], [8] by IBM, have been proposed. 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.

For these reasons, some researchers have started to realize the importance of using discrete and declarative constraints for system management [9], although most of their research is at an early stage and concerns the propositions of concepts or architecture, in contrast with our approach, which is based on sound logical and mathematical foundations.

From the viewpoint of algorithms, state search algorithms such as A*-algorithm [24] and integer linear programming [25] have been widely used for planning problems with constraint satisfactions. These algorithms usually require defining the complete description of their goal state to be achieved. However, in the complex IT system management, we usually know only a fraction of all constraints to be satisfied in configuration procedures and goal states. Therefore, proper configuration procedures to achieve the goal condition with satisfying given constraints can vary depending on the initial system configuration. This makes it difficult to apply common planning algorithms directly to the configuration procedure synthesis problems.

Some existing research approaches have applied Alloy for system configurations [16], [17], [26]. These approaches have aimed at finding correct static configurations at specific points in time or detecting possible constraint violations. They are therefore completely different from our approach, which synthesizes the procedure for dynamically changing system configurations using discrete knowledge.

3. Alloy and Alloy Analyzer

Alloy is a specification language developed at MIT based on set theory and first-order logic. The Alloy Analyzer [13] can check the satisfiability of a set of formulas written in Alloy 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 [14] written in Java and miniSat [15]. 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.”

There are three main reasons for using Alloy and the Alloy Analyzer in our method of synthesizing the configuration procedure.

- 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. Consequently, it is suitable for modeling today’s IT 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.

4. 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.
4.1 Architecture

The high-level architecture in our method is depicted in Fig. 1. 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) [10] — a storage solution for managing relationships between IT components that was proposed in the Information Technology Infrastructure Library (ITIL) [11]. We implemented the CMDB on the AXIS2 [21] server and used Resource Control eXtensible Markup Language (RCXML) [12] for the data format. RCXML is a customized XML format used to integrate system and management information.

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 define 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. Here we suppose that each expert has complete knowledge regarding the pre- and post-condition of each atomic operation. In other words, we suppose that there is no side effect of an operation which is not recognized by the expert. We also suppose that the experts have enough knowledge regarding first-order logic to be able to describe their knowledge on system management in Alloy language.

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.

4.2 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 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 Fig. 2.

1. System structure
The system we considered in our case study (shown in Fig. 2)
in Fig. 2) 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 [18] 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 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 [20] services, the smallest set of resources provided by the service consists of one processor (Intel Xeon or AMD Opteron) with a 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 \( a \) (a parameter in the case study) 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 Fig. 3. This figure shows the physical configuration of Server_A consisting of three physical components (the CPU, memory, and hard disk) by connecting the elements representing these components with the Server_A element via “componentOf” links. Figure 4 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 5 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 (the 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 -> Server_A) and (Memory_A -> 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.

(2) Executable operations
We assumed that four operations could be executed in the system.

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 can determine changes in configurations triggered by these operations. Figure 6 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 $\text{connect}(src,dst)$ shows the following three facts hold.

(A) No $\text{connectedTo}$ relation from $src$ to $dst$ before connect operation is executed
(B) No $\text{connectedTo}$ relation from $dst$ to $src$ before connect operation is executed
(C) The set of $\text{connectedTo}$ relations after the connect operation is the union of the set of ones before the operation and $\text{(src -> dst)}$

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 [23] 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 $s'.\text{changes}=*$ added to the last part of each operation definition; the relations between the flags and the system status changes are defined in Fig. 7. For example, in the definition of the connection operation in Fig. 6, the description, $s'.\text{changes} = \text{connectedTo}_c$, clarifies that the operation only changes $\text{connectedTo}$ relations by including the flag, $\text{connectedTo}_c$, in the set of flags $s'.\text{changes}$. At the same time, the frame condition, "$s.\text{connectedTo} = s'.\text{connectedTo} || \text{connectedTo}_c \in s'.\text{changes}$", described in Fig. 7 defines that unless this flag is included in the set $s'.\text{changes}$, the $\text{connectedTo}$ 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 Fig. 8. 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’.

(3) 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.

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 Fig. 9. 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 Fig. 9.

(4) 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 sufficient capacities to accommodate them. If α is more than 2, on the other hand, we cannot consolidate them into Server_A due to capacity constraints. 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...
Fig. 10  Goal condition.

Fig. 11  State search with intermediate states.

4.3 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 model finder. Then Alloy 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 representing the transitive closure of relations without any reflexive relations.

(1) Identify the goal and intermediate states
Here we first derive “concrete goal conditions” by assigning possible values for variables $X = (x_1, x_2, \ldots, x_m)$ in the set of predicates $Pred = \{pred_1, pred_2, \ldots, pred_n\}$ which are to be satisfied to achieve goal condition. We represent the possible value assignments for $x_i$ by $A(x_i) = \{C_{i1}, C_{i2}, \ldots, C_{il}\}$ and all combination of the value assignments by $P = \{p_1, p_2, \ldots, p_L\} = A(x_1) \times A(x_2) \times \cdots \times A(x_m)$. We represent the predicate $pred_i$ with an assignment $p_j$ by $pred_i^{p_j}$. We define the set of concrete goal conditions $CGC = \{cgc_{i1}, cgc_{i2}, \ldots, cgc_{ik}\}$ where $cgc_i = \{pred_1^{p_j}, pred_2^{p_j}, \ldots, pred_n^{p_j}\}$.

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. $L: S \rightarrow 2^{pred} \times P$ 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 $S_G = \{s_{G1}, s_{G2}, \ldots, s_{Gk}\} \subseteq S$ satisfying the concrete goal condition $(L(s_{Gi}) = cgc_i)$. Then we define intermediate states $s_{(i,j)} \in S$ with $L(s_{(i,j)}) = \{pred_k^{p_j} \mid k \leq j\}$ representing the states where $j$ out of $n$ predicates in cgc_i are satisfied under the assumption that each predicate is independent of the other predicates and the order of predicates to be satisfied does not affect the reachability to the goal state. By defining the transition relation so that the predicates satisfied in the states increment one by one along with the transition path, we complete the identification of the states to be searched $(s_{(i,j)}, s_{(i,j+1)}) \in T \text{ if } L(s_{(i,j)}) = L(s_{(i,j+1)}) \cup \{pred_j^{p_j}\}$.

As a result of this step, the path to be investigated described in Fig. 11 (a) can be modified as shown in Fig. 11 (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 oper-
condition. When VMM C is achieved, OS A1 and Guest X are consolidated into VLAN V1. A2 on which OS A2 is migrated from 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 ofGuest_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 use 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 Fig. 12. When α = 2, the output result (Fig. 12 (a)) indicates that we can achieve the required configuration in just one step by migrating Guest_OS_B1 from VMM_B to VMM_A. However when α = 4, the output (Fig. 12 (b)) indicates that we need two steps to migrate 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 to the capacity constraints. Then when α = 8, the procedure consisting of the following six-step operations is obtained (Fig. 12 (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. Migrate Guest_OS_A1 to VMM_C.
5. Migrate Guest_OS_A2 to VMM_C.
6. Migrate Guest_OS_B1 to VMM_C.

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

6. 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, 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 256 Mbyte, 512 Mbyte and 1024 Mbyte 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 Fig. 13(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 5 steps in the procedure synthesis for each transition to the goal state \( s_G \) as shown in Fig. 13(b). In this evaluation, we used the three predicates \( \text{Pred} = (x \in \text{Web}_X. \text{runningOn}, x \in \text{App}_X. \text{runningOn}, x \in \text{DB}_X. \text{runningOn}) \) to be satisfied in the goal state and constructed the intermediate states along with the order of the predicates in \( \text{Pred} \).

In this case study, since there is no pre-/post-condition relation between these predicates, the order of these predicates in \( \text{Pred} \) 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 2 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, 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 analy-
sis 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 investigated in our approach as shown in Fig. 13 (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 (increase by 12%).

7. 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 Sect. 6, 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 approaches to achieve this. For example, we can apply some effective CNF construction techniques proposed in [22]. It will also be possible to reduce the size of system model by abstracting some parts irrelevant to the purpose of system configuration changes.

8. Conclusion

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. We demonstrated that our method could derive appropriate procedures for different configurations and evaluated the memory consumption through a case study. 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 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 improve-
ment. For example, in our case study scenario, we suppose that the order of predicates to be satisfied in each intermediate goal does not affect the reachability to the goal state. However, it is still possible that some efficient heuristics can enable us to find better order of the predicates by which shorter procedure can be synthesized. By combining techniques for efficient intermediate goal selection with a model finding technique that can check the satisfiability of formulas consisting of over 10 million SAT clauses, it would be possible to overcome scalability problems in today’s complex large-scale IT systems.

Acknowledgments

The authors would like to thank Dr. Masazumi Matsubara for his useful advice on CMDB and RCXML, and thank Dr. Radu Calinescu for providing us with many useful comments on our manuscripts.

References


Shinji Kikuchi is a senior researcher at FUJITSU LABORATORIES LTD., and a Ph.D. student at Japan Advanced Institute of Science and Technology (JAIST). His research interest includes cloud computing, autonomic computing, ICT system management, formal methods and verification. Prior to joining Fujitsu Laboratories, he received B.E. and M.S. degrees in Information technologies from Nageya University. He is a member of IEEE and has been on the program committees of multiple international conferences on autonomic computing, complex information systems and cloud computing.

Satoshi Tsuchiya received from Tohoku University M. E. degree in 1991. He is currently a research manager at Fujitsu Laboratories. His research interests include cloud computing, parallel processing of big data.

Kunihiko Hiraishi received from the Tokyo Institute of Technology the B. E. degree in 1983, the M. E. degree in 1985, and D. E. degree in 1990. He is currently a professor at School of Information Science, Japan Advanced Institute of Science and Technology. His research interests include formal modeling of systems, discrete event systems and formal verification. He is a member of the IEEE, IPSJ, and SICE.