A Study on the Intent-based System Design Automation Method for Fault Detection and Fault Tolerance

Takayuki Kuroda, Toshiki Watanabe and Tatsuya Fukuda

NEC Corporation, Secure System Platform Laboratories, Kawasaki, Kanagawa, 211-8666, Japan
kuroda@nec.com, tos_nabe@nec.com, t_fukuda1987@nec.com

ABSTRACT

ICT systems are becoming an important infrastructure indispensable for business operations in various industries. While the requirements are becoming more diverse and complex, stable service provision is required. As a result, the burden on operations is increasing, and there are growing expectations for automation technology to solve this problem. This study investigates techniques to significantly reduce the burden on operators by automating the construction and maintenance of systems that satisfy Intent, a concise description of users' requirements. However, even using the results of previous research, it has been difficult to automate operations with consideration for the stability of service provision. Therefore, this paper describes an automated method based on intent for detecting failures that occur in the target system and an automated design method for fault-tolerant systems. First, we describe the basic concept, present the results of thought experiments, and discuss its effectiveness.

1. INTRODUCTION

In recent years, ICT systems have become critical infrastructure supporting business operations in many industries, and their prompt and stable provision has become an increasingly important issue. At the same time, the diversification of needs for ICT systems and the evolution of virtualization technologies have made system configurations more complex, and stable operation has become more difficult. Therefore, automation technology is needed to operate systems easily and stably (Beyond 5G Promotion Consortium White Paper Subcommittee, 2023).

In response to these needs, international standards organizations such as TMForum and ETSI have proposed the concept of intent-based autonomous network operation (TM Forum, 2020)(ETSI, 2022). In autonomous operation, the user only needs to enter the intent, and the network that satisfies the intent is expected to be automatically constructed and maintained. The author's group has also promoted research and development of autonomous operation of ICT systems, and in particular, has proposed an automatic design technique for ICT systems based on intent (Kuroda, 2022).

However, existing automatic design based on intent has neither established a design method that takes fault tolerance into account nor a fault detection method, making it difficult to use for stable system operation. Therefore, this study proposes an extension of automatic design techniques to include fault-tolerance considerations and a fault detection mechanism. This paper outlines the basic methodology, presents the results of a thought experiment on its operation based on a sample scenario, and discusses its effectiveness.

In the following sections of this paper, Section 2 provides an overview of existing automatic design techniques, Section 3 describes the proposed automatic design method with fault detection and fault tolerance, and Section 4 presents thought experiments and discusses its effectiveness. Finally, Section 4 presents the conclusions.

2. AUTOMATED SYSTEM DESIGN TECHNOLOGY

Automated system design technology is a technology that derives and outputs a concrete system configuration upon input of an intent describing abstract system requirements. Because automated design technology is a technology that

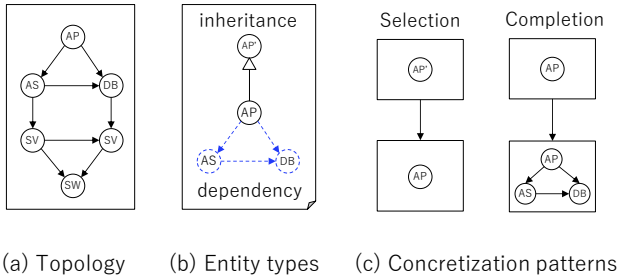

Figure 1. Main data handled by automated system design technology.

mechanically mimics the thought process of human design, it is easy to understand the mechanism for automated design if we first imagine the thought process of a human designer. When humans design a system, they are first given abstract requirements. The requirements consist of several functional and non-functional requirements. The designer replaces the functional requirements with equipment that correspond to the specific means of achieving them. There are often multiple choices of equipment that can be selected, and the designer selects the equipment that will meet the given non-functional requirements. The selected equipment also has other requirements, such as other functions or a certain amount of resources, as constraints for it to operate properly. The designer searches for a combination of equipment that will satisfy all of these conditions. In order to make the search process efficiently, the designer should not suddenly select a specific product, but rather concretize the process step by step, first establishing categories of realization means, and then selecting specific products if they are appropriate.

Below is an overview of the main data and algorithms handled by automated system design technology.

Figure 1 shows the main data handled by Automated system design technology. The input intent, the output configuration, and the proposed configuration that appears during the design process are all represented as a topology in the form of a graph, as shown in Figure 1 (a). Nodes represent components, and edges represent relationships between components, which together are called entities. Entity must have id and type and may have properties and constraints. The abbreviations in the figure have the following meanings: AP means Application, AS means Application Server, DB means Database, SV means Server and SW means Network Switch.

Figure 1 (b) shows the definition of the type of an entity. type mainly defines inheritance, dependency, and a flag indicating whether it is a concrete type or not. Inheritance indicates what kind of type it is, and dependency indicates the expected surrounding topology required for an entity of that type to operate. For example, Linux inherits the OS and has as a dependency the surrounding topology of being hosted by Machine.

There are two types of topology embodiments: *selection* and *completion*. This is illustrated in Figure 1 (c). *Selection* is the operation of replacing one abstract entity type with another that inherits it. The type to inherit is given by the definition of inheritance. *Completion* is the operation of adding a surrounding topology around an entity, the specific method of which is derived from the definition of dependency.

Figure 2 shows the algorithm for automated system design. Automated system design is a search process in which, starting from an intent, the steps of concretization consisting

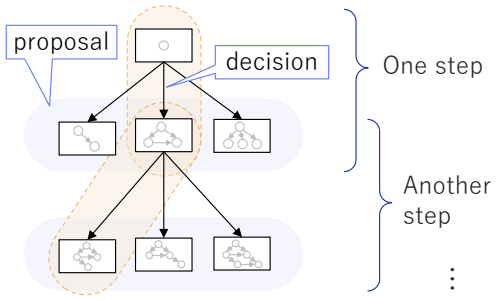

Figure 2. Algorithm of Automated System Design.

of *proposals* and *decisions* are repeated, such that the end condition is the discovery of a *fully concretized* topology. *Proposal* is a process to generate multiple new topologies that are one level of embodiment of a certain topology. New topologies are generated by applying possible candidates for *selection* and *completion* to each entity in the target topology. *Decision* is the process of selecting the most promising one from the new generated topologies. It verifies that each topology satisfies the constraints and evaluates the optimization conditions to determine the topology that satisfies the conditions and has the highest evaluation. A Topology is considered *fully concretized* when all the entity types it contains are concrete and dependencies are satisfied.

3. Fault Detection and Fault Tolerance with the Intent-Based System Design Method

This section describes in detail the fault detection method assuming the use of the automated system design and the fault-tolerant automated system design technique. The effectiveness of this method is also discussed after showing the results of thought experiments.

3.1. Fault detection using dependencies between entities

As described in Section 2, automated system design repeatedly complements dependencies to reach the design result. Therefore, in the topology of the design result, the

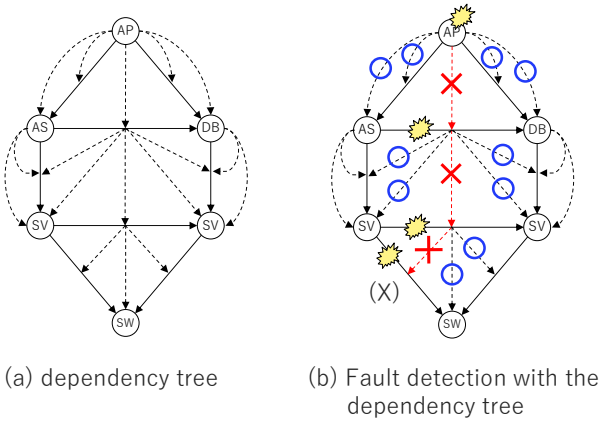

Figure 3. Fault detection with dependency tree.

dependencies among all entities are clarified. By tracing these dependencies, faults can be detected. Figure 3 shows a dependency tree and fault detection using a dependency tree. As shown in Figure 3 (a), dependencies are recorded between each entity in the topology of the design result, forming a tree structure. Figure 3 (b) shows the fault detection using the dependency tree when a functional failure of the AP is observed. First, the state of each entity on which the AP depends is examined, and it is found that the connection between the AS and the DB has become disconnected. Then, by examining the status of each entity on which the connection between AS and DB depends, we can see that the connection between the servers (SV) that host them is disconnected. Finally, by checking the status of entities on which the connection between SVs depends, the root cause can be found: the wired connection between the server hosting the AS and the SW is broken. By defining the method of checking the status of each entity in advance for each entity type, a series of status check processes can be automated.

By traversing the dependency tree in the reverse direction, it is also possible to check the range of influence when the function of a certain component is stopped. For example, if the SW in the figure is to be stopped for maintenance, it can be confirmed that stopping the SW will affect the function of the AP.

3.2. Automated system design for fault tolerance

3.2.1. Overview

Next, we discuss automated system design with fault-tolerance in mind. First, the definition of fault-tolerance in this paper is presented, followed by an explanation of the mechanism for automated design with such properties.

In this paper, we define that an entity *e* has fault tolerance *n* as the fact that *e* does not malfunction even if *n* of the entities on which *e* depends malfunction, and we denote this as *isTolerant(e, n)*. In other words, if *n* is 1, it is tolerant to a single point failure, and if *n* is 2, it is tolerant to a two-point failure. In practice, it is known that tolerance to two-point failures is sufficient, and in this study, the value range of *n* is considered to be from 0 to 2.

Next, we discuss fault-tolerant automated system design. In general, extending automated system design to design a topology with certain properties requires two things: (1) extending the proposal mechanism for automated design to derive the desired topology candidates, and (2) extending the decision mechanism to select the one with the desired properties from the generated topology candidates. In the sections that follow, we describe the concretization patterns for proposing candidate redundant topologies and constraints on fault-tolerance and provide examples of design results obtained through automated system design with these patterns and constraints.

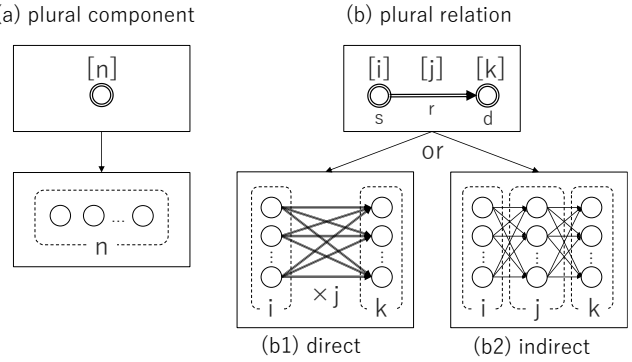

Figure 4. Concretization patterns for plural entities.

3.2.2. Concretization patterns that generate redundant topologies

To ensure that an entity *e* does not become malfunctioning even if some of the entities on which *e* depends are malfunctioning, the entities on which *e* depends must be sufficiently redundant. Therefore, we first distinguish between singular and plural forms of an entity and assign quantity as an attribute to express the number of pieces. Then, we introduce a concretization pattern such that a plural entity is decomposed into its number of singular entities.

Figure 4 shows concretization patterns for plural entities. Figure 4 (a) is a pattern that decomposes a plural component, and Figure 4 (b) is a pattern that decomposes a plural relation. Entities drawn with double lines indicate that they are plural entities. The subscripts on the plural entities indicate the quantity of the entities. The dotted rounded rectangles and numbers on the decomposed topology indicate the number of singular entities that have been decomposed from the same plural entity. The decomposition of component is relatively simple. As shown in Figure 4 (a), *n* plural components are decomposed into *n* singular components. Compared to component, the decomposition of relation is somewhat more complex. As shown in Figure 4 (b), there are two ways to decompose a plural relation: by directly connecting it to the entities at both ends (Figure 4 (b1)) or by relaying other entities (Figure 4 (b2)). First, let us discuss the former method of direct connection. Forgetting for the moment that relation is plural, let us consider the decomposition of the components at both ends of relation. In Figure 4 (b1), *s* and *d*, the components at both ends of relation *r*, are plural components with *i* and *k* quantities, respectively, and are decomposed into *i* singular components of s_1, s_2, … s_i, and *k* components of d_1, d_2, ... d_k, respectively. At this point, *r* is understood to be the mesh that connects these $s_{1..i}$ and $d_{1..k}$, respectively. In addition to the above, the fact that *r* is plural means that each connection between $s_{1..i}$ and $d_{1..k}$, is plural. In Figure 4 (b1), each relation is drawn as a triple line and $\times j$ indicates that there are *j* connections each. In Figure 4 (b2), *j* relay entities

if e is	singular	plural
composite	forall(e.depends, (i)->{isTolerant(i, n)})	e.quantity > n && exclusive(e)
primitive	n == 0	

Figure 5. Condition for function isTolerant(e, n) to return true.

are placed between *s* and *d*, each of which is mesh connected. In this case, each relation is singular. Figure 4 (b1) and (b2) both show that the relation between any $s_{1..i}$ and $d_{1..k}$ is preserved even if multiple connections or relay entities are broken.

Using the above patterns, we can propose any necessary and sufficient redundant topologies by assuming that the quantities of the entities can be 0, 1, and 2.

3.2.3. Conditions for fault-tolerant topologies

We describe conditions for determining suitable fault-tolerant topology candidates from the redundant topologies generated by the methods described in the previous section. Figure 5 shows the condition for function isTolerant(e, n) to return true. Here e is the entity to be evaluated for fault-tolerance, and n is the maximum number of failure points at which malfunctioning of e should be avoided.

The conditions are divided according to whether the type of entity e is singular or plural, and whether it is composite or primitive.

Here, we introduce the new concepts of composite and primitive. composite is a conceptual entity that represents a group of multiple entities. For example, the system shown in Figure 7 refers to the application, the database, and the connection between them, and is a grouping of these three entities. Primitive is an entity that has its own substance. When a Composite is concretized, the entities it contains are added to the surrounding topology. The embodiment of Composite is similar to the completion in Figure 1(c), but the meaning is somewhat different. The latter is an operation that adds other dependent entities, while the former is an operation that makes explicit the entities it encompasses. The impact of a fault-tolerance requirement for a primitive entity on the entities it depends is different from the impact of a fault-tolerance requirement for a composite entity on the entities it contains.

Here, the condition for function *isTolerant(e, n)* to return true is that:

- if *e* is singular, it depends on whether it is composite or primitive:
  - if *e* is composite, each entity *i* it contains must satisfy *isTorelant(i, n)*;
  - if *e* is primitive, *n* must be *zero*.
- If *e* is plural, then the quantity of *e* is greater than *n* and the entity *e* depends on satisfies the *exclusivity condition* described below.

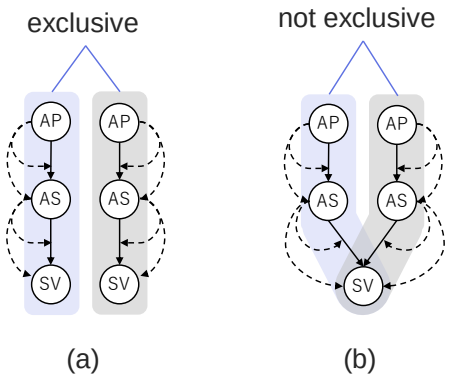

Figure 6. Exclusiveness of dependency.

Figure 6 illustrates the exclusivity of dependent entities. Two entities are mutually exclusive if the topologies on which they depend do not contain any identical entities. Let us assume that the dependent topology of an entity is the topology consisting of the entities on which it depends and the group of entities obtained by iteratively tracing the entities on which it further depends. Figure 6 (a) is mutually exclusive, while (b) is not exclusive because the two dependent topologies contain the same entity of SV.
By checking for exclusivity, fault tolerance can be assured. For example, even if redundancy is used to provide fault tolerance for an application, if those applications are deployed on a single server, that server will become a single point of failure. By checking for exclusivity, it is possible to guarantee that each server, the entity on which an application depends, is provided with its own server, thereby avoiding the occurrence of a single point of failure.
However, since too much exclusivity can lead to excessive redundancy, it is effective to add measures to mitigate the strictness, for example, by assuming that the cloud infrastructure itself is highly reliable, so that even if two entities contain the identical entities of the cloud infrastructure, the exclusivity between those entities is not impeded.

3.2.4. Automated design for fault tolerance

The basics of automated system design, concretization patterns for generating redundant topology candidates, and fault-tolerance conditions have been described so far. Fault-tolerant automated system design is achieved by repeating concretization steps in which redundant topologies are proposed and a proposal that satisfies the fault-tolerant condition is selected from among the redundant topologies in each concretization step.

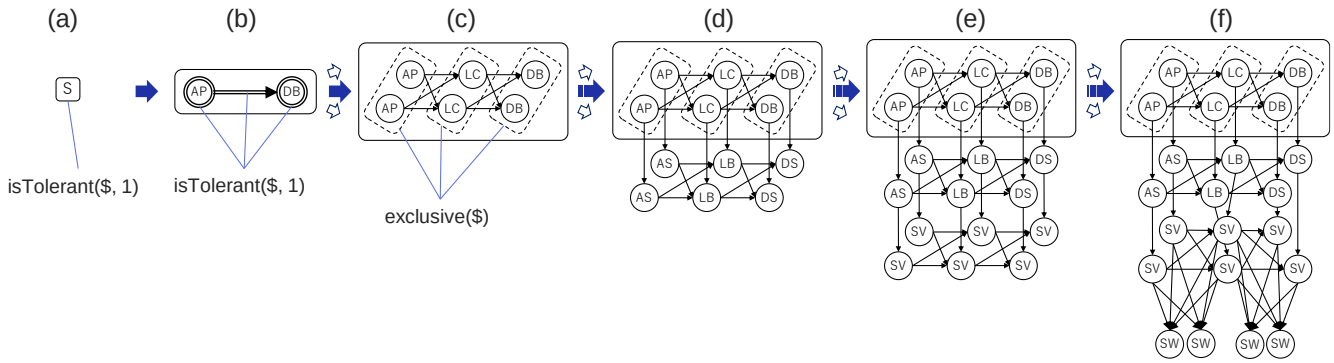

Figure 7. Example of automated system design with a constraint of fault tolerance.

Figure 7 is an example of a thought experiment in automated system design with redundancy. (a) is the input intent. This intent defines one component of the system type and specifies fault tolerance against single point failure as a constraint condition. The "$" in the figure means that the entity to be constrained, indicated by the line, is assigned. In (b), the system is expanded to introduce the internal application (AP) and database (DB) and their connection relations. As already mentioned, the condition for an entity of Composite type to be fault-tolerant is that all its internal entities are fault-tolerant, so the constraint condition is delegated to each internal entity. The entities in the system are all plural entities, because they were specified as plural in the definition of the system type. If "singular" is specified here, no candidate topology that satisfies fault-tolerance can be generated at this point, and the design will fail. In (c), each plural entity is decomposed into two entities and given an exclusivity condition, and the connection between the application and database is converted to a connection through the load balancer's configuration (LC). Although the configuration in which each entity is decomposed into three or more entities also satisfies the condition of fault tolerance, the minimum necessary configuration is selected by taking other conditions such as resource saving into account. In the following (d) and (e), the dependencies of each entity are complemented. In order to satisfy the exclusivity condition, a topology has been selected in which different entities are assigned to each of them. In (f), all dependencies have been complemented, so the design is completed here in this thought experiment. In practice, concretization will continue until a specific product for each entity is selected.

3.3. Discussion

For human engineers in general, designing ICT systems that meet fault tolerance requirements is a complex and difficult task. In particular, the task of safely performing complex operations, such as changing the requirements of a system in operation or temporarily changing some configurations for maintenance, while maintaining fault tolerance is very complex and carries the risk of causing failures.

Using this technology, even complex operations such as those described above can be performed safely by simply and reliably designing configurations that meet the desired fault tolerance.

In addition, along with fault tolerance, it is possible to combine performance, budget, and other requirements to design an efficient and secure configuration with the necessary and sufficient redundancy in the right place at the right time.

In addition, failure detection using dependencies between entities, which are clarified at design time, can be used to automatically check whether the required redundancy is being met.

4. Conclusion

This paper describes an extension to consider fault tolerance in automated system design techniques and a fault detection method that uses information on dependencies among entities that arise in the automated design process. After presenting the design results of the thought experiment, the paper shows that the technique can easily and reliably perform complex system design tasks to satisfy fault-tolerance requirements. In the future, we will demonstrate the effectiveness of this technique through a prototype of an automated system design function based on the method described in this paper.

References

Beyond 5G Promotion Consortium White Paper Subcommittee, "Beyond 5G White Paper - Message to the 2030s -", version 2.0, March 13, 2023. https://b5g.jp/doc/whitepaper_en_2-0.pdf

TM Forum. Autonomous Networks: Empowering Digital Transformation For Smart Societies and Industries. TMForum White Paper, 2020. https://www.tmforum.org/resources/whitepapers/autonomous-networks-empowering-digitaltransformation-for-smart-societies-andindustries/.

ETSI, “Intent driven management services for mobile networks”, TS 128 312 V17.0.1 (3GPP TS 28.312 version 17.0.1 Release 17), Jul. 2022.

T. Kuroda, Y. Yakuwa, T. Maruyama, T. Kuwahara and K. Satoda, "Automation of Intent-based Service Operation with Models and AI/ML," NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, 2022, pp. 1-6, doi: 10.1109/NOMS54207.2022.9789924.

Takayuki Kuroda received M.E and Ph.D. degreed from the Graduate School of Information Science, Tohoku University, Sendai, Japan in 2006 and 2009. He joined NEC Corporation in 2009 and has been engaged in research on model-based system management for Cloud application and Software-defined networks. As a visiting scalar in the Department of Electrical Engineering and Computer Science at the Vanderbilt University in Nashville, he studied declarative approach of automatic workflow generation for ICT system updates from 2013 to 2014. He is currently working at NEC on automation technology for system design, optimization, and operation.

Toshiki Watanabe received Ph.D. from the Graduate School of Information Science and Technology, Osaka University, Suita, Japan in 2010. He joined NEC Corporation in 2010 and has been engaged in research on SDN (Software-Defined Networking). His research interests include security and designing network architecture.

Tatsuya Fukuda received Ph.D. degreed from the Graduate School of Information Science and Technology, Osaka University, Suita, Japan in 2015. He joined NEC Corporation in 2015 and has been engaged in research on network performance measurement and estimation. He is currently working at NEC on automation technology for system design, optimization and operation.