Mutation Analysis for System of Systems Policy Testing

Wonkyung Yun, Donghwan Shin, Doo-Hwan Bae
School of Computing
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon, South Korea
{wkyun, donghwan, bae}@se.kaist.ac.kr

Abstract*—A System of Systems (SoS) is a set of the constituent systems (CS) which has managerial and operational independence. To address an SoS-level goal that cannot be satisfied by each CS, an SoS policy guides or forces the CSs to collaborate with each other. If there is a fault in the SoS policy, SoS may fail to reach its goal, even if there is no fault in the CSs. Such a call for SoS policy testing leads to an essential question—*how can testers evaluate the effectiveness of test cases?

In this paper, we suggest a mutation analysis approach for SoS policy testing. Mutation analysis is a systematic way of evaluating test cases using artificial faults called mutants. As a general mutation framework for SoS policy testing, we present an overview of mutation analysis in SoS policy testing as well as the key aspects that must be defined in practice. To demonstrate the applicability of the proposed approach, we provide a case study using a traffic management SoS with the Simulation of Urban Mobility (SUMO) simulator. The results show that the mutation analysis is effective at evaluating fault detection effectiveness of test cases for SoS policies at a reasonable cost.

***Keywords*-System of Systems, Mutation Analysis, Simulation of Urban MObility (SUMO), System of Systems Policy, System of Systems Testing**

I. INTRODUCTION

In recent years, unsolved problems within a single system have grown rapidly along with software and system complexity. In order to address such problems, existing systems can be integrated to become a more complex and larger system, incurring more complexity in the integrated systems. This is called a System of Systems (SoS) [1], which is a collection of Constituent Systems (CS). CSs are expected to cooperate with each other in order to achieve an SoS-level goal; each CS also operates independently in order to achieve its own goals. To guide how CSs cooperate with each other as an SoS, in response to changing environments, there are predefined rules called SoS policies [2], [3], [4]. An SoS policy is a means used to influence a system in order to achieve its goals [5]. For instance, according to the SoS policy, a manager of a disaster response SoS can guide or order ambulances and fire engines to rescue patients from the conflagration together. Interestingly, the SoS-level goal could fail not because of faults in the CSs but because of faults in the SoS policy. In this regard, SoS policy testing is important in SoS engineering. However, there is no way to evaluate the quality of test cases for SoS policy testing.

To address this problem, we propose mutation analysis for SoS policies to evaluate the fault detection effectiveness of SoS policy test cases. In the detection of real faults, mutation criteria have been considered more effective than structural coverage [6], [7]. In mutation analysis for general testing, many artificial faults (i.e., mutants) are generated from an original program and used to assess the fault detection effectiveness of the test cases. However, adapting the notion of mutation in SoS policy testing is not straightforward. For example, in SoS policy testing, we must define what a mutant is and what it means to kill mutants. Thus, we provide an approach and also a case study of an SoS scenario for a traffic congestion control, which has been used in several studies [8], [9], [10], to demonstrate the effectiveness and time complexity of mutation analysis for SoS policy testing. To execute SoS policies in a traffic congestion control SoS, we use an open-source simulation engine called Simulation of Urban MObility (SUMO) [11].

This paper is organized as follows. Section II introduces related work on SoS policy and SoS testing. Section III describes the basic concepts and the processes of mutation analysis and SUMO. In Section IV, we propose mutation analysis for SoS policy testing. Section V explains experimental subjects and Section VI shows the effectiveness and the time complexity of the proposed mutation analysis approach. Section VII concludes the paper.

II. RELATED WORK

Testing a policy in SoS is very important in that SoS may fail even if its CSs do not have faults in them. However, to our knowledge, SoS policy testing has not been studied.

Hall-May et al. defined an SoS policy as a set of rules to control the behavior of an SoS in order for it to appropriately operate in a given environment [2]. DeLaurentis et al. suggested an SoS lexicon with respect to categories and levels, and applied this lexicon to analyze the National Transportation System [3]. According to them, an SoS policy is a function that can guide the entities of the SoS. Based on the view of Walker [12], Agusdinata stated that a policy is a means to influence the system in order to achieve certain goals [5]. Later, Agusdinata et al. presented an approach to the design of an adaptive SoS policy that would consider uncertainty and complexity [13]. Mostafavi et al. recently

proposed an ex-ante policy analysis framework by combining an agent-based modeling approach and system dynamics [14], [15]. They considered an SoS policy to establish a sustainable infrastructure on changing environment.

In SoS testing, Luna et al. [16] suggested a framework for Integration, Verification, Validation, Test, and Evaluation of SoS. They combined several existing approaches: the Department of Defense Architecture Framework to identify SoS, the Decision Structure Matrix to optimize the interactions between CSs, and the pairwise testing technique to determine the minimum test set. In order to overcome complexity and cost issues, Liang et al. suggested a framework for SoS testing using randomization theory to prioritize test cases [17]. Nielsen et al. categorized SoS testing challenges into areas of complexity, management, multiple standards, and issues related to the dynamic evolution of the SoS configuration [10]. They insisted model-based testing as one solutions [10], [18] because this solution can provide a capability view of SoS and the possibility of extracting test cases.

In addition to testing an SoS itself, we believe that it is also very important to test the SoS policy. In SoS, the role of the SoS policy becomes more significant because individual CSs are not designed to support SoS-level goals. Our work aims at evaluating the fault detection effectiveness of test cases for SoS policies, and provides guidelines to manage and operate CSs.

III. Background

A. Mutation Testing

Mutation testing is a fault-based testing technique proposed in 1970s by Lipton [19] and developed by DeMillo [20]. It originated from the idea that if a test case can detect an artificially seeded fault, the test case also can detect a real fault. In mutation testing, a program that has a seeded fault is called a “mutant", and the rules of injecting faults are called mutation operators. If the execution result of a mutant is different from that of the original program for a test case, it is said that the mutant is killed by the test case. The ratio of mutants killed for a set of test cases is called the mutation score of the set of test cases. This situation has been widely studied and it has been stated that making higher mutation scores is effective at detecting real faults [7], [21]. In this paper, we apply a mutation criterion in order to evaluate test cases for SoS policy.

B. Simulation of Urban MObility

Simulation of Urban MObility (SUMO) is an open-source modeling and simulation tool for the transportation domain [11]. It was developed by the Institute of Transportation Systems at the German Aerospace Center and has been consistently updated so far since 2000. With SUMO, users can visually see the simulation results and conduct experiments using algorithms related to traffic, such as routing. An example of the simulation is shown in Figure 1.

In order to simulate SUMO, it is necessary to include the map (roads and intersections) and vehicle (types, routes and

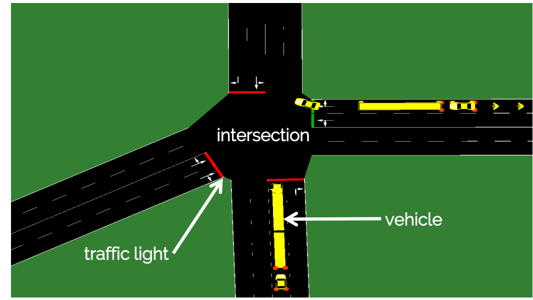

Fig. 1. An example of SUMO simulation. SUMO provides the visual simulation of traffic flow.

instances) information. Map information involves the details of the roads and intersections, such as the length of roads and their location. Vehicle information includes vehicle types and routes, and generate vehicle instances. Vehicle type can be comprised of max speed, acceleration, deceleration, length of types, etc.; route specifies where the vehicle leaves from and the destination. A vehicle instance is generated by assigning a vehicle type and a route. The users can provide supplemental information such as traffic lights. Since transportation SoS have been widely studied, we modeled a traffic congestion control SoS as an example scenario through SUMO.

IV. Mutation Analysis for SoS Policy Testing

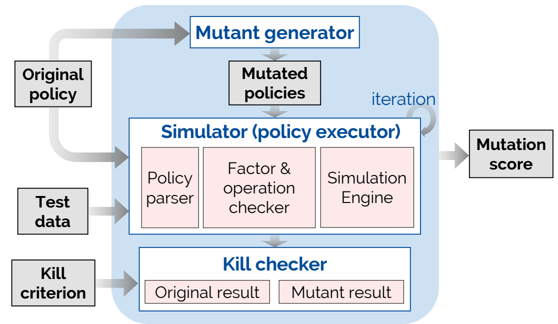

Fig. 2. Overview of mutation analysis for SoS Policy Testing

The overall mutation analysis approach for SoS policy testing is shown in Figure 2. The approach is composed of three key parts: a mutant generator, a simulator, and a kill checker.

- The mutant generator generates many mutants from an original SoS policy by applying mutation operators.
- The simulator simulates (or executes) the generated mutants and the original SoS policy for the given test cases. To maintain our mutation analysis approach general, we keep the simulation engine as abstract as possible.
- The kill checker uses the kill criterion to determine whether individual mutants are killed by the test cases or not. It finally returns the mutation score for the test cases, indicating the fault detection effectiveness of the test cases. A set of test cases is said to be more effective

if it kills more mutants, which can be represented as a higher mutation score.

In the following subsections, we explain the essential aspects of the proposed mutation analysis approach for SoS policy testing.

A. SoS Policy and its Structure

Although the SoS policy has been studied in different ways by several researchers, all researchers to this point have essentially thought of the SoS policy as a set of rules to guide and influence the SoS in achieving goals [2], [3], [4]. By deriving from the definition of SoS policy in [4], we define a basic SoS policy structure as pairs of factors and operations. The operation part refers to how an SoS policy affects an SoS, including the CSs. The factor part refers to times at which the operation should be activated in terms of the SoS environment. Each part must have elements to perform its roles, as shown in Table I.

TABLE I
STRUCTURE OF AN SOS POLICY

Category	Elements
Factor	Monitoring CSs
	Monitoring targets
	Satisfying states
Operation	Operating CSs
	Description of functions (e.g. function name, sustaining time)

The first element of a factor is a set of monitoring CSs which refers to what CSs monitor the environment of SoS. Since each CS is capable of monitoring multiple environmental targets, specific monitoring targets should be described in the factor as well. Finally, the satisfying state specifies in what state the factor is determined to be satisfied. For example, consider a traffic control SoS. Let each CS is composed of the traffic monitoring cameras as sensors and the traffic control lights as actuators in each direction, and placed at each intersection. The factor could be described in such a way that *the number of vehicles waiting for signal in one direction exceeds ten vehicles*. If the factor is satisfied, a corresponding operation should be triggered. The operation should include which CSs will operate (i.e., operating CSs), and how they operate (i.e., description of functions). In above example, the corresponding operation could be described in such a way that *turns the signal in the direction in which the factor is satisfied into blue light*. Depending on the domain and the capabilities of the SoS, detailed requisites in the operations and factors can be slightly different.

B. Mutant Generator

Once an SoS policy is defined, it is mutated using mutation operators for testing. This mutation process is conducted on a mutant generator. A mutation operator is a rule that makes syntactic changes in a target program. Since there is no mutation operator for an SoS policy, we develop 8 mutation operators for the SoS policy with reference to the mutation operators for Fortran and Simulink [22], [23]. Mutation operators and corresponding descriptions are shown in Table II. For instance on the example of Section IV-A, a mutant generated by ROR mutation operator can turn the signal intro blue light fan when the number of vehicles waiting for the signal is *less than* ten, instead of *more than* ten. In Section VI-B, we evaluate whether the mutants generated by the mutation operators for the SoS policy can mimic real faults.

C. Simulator

After mutants are generated, the mutants and the original SoS policy must be executed for the test cases. The policy parser converts the format of the SoS policies in order to execute them in the simulation engine. When the policy parser returns the parsed pairs of factors and operations, the factor and operation checker repeatedly verifies whether each factor is satisfied during the whole simulation. When some factors are satisfied, the corresponding operations are marked as executed by the simulation engine. Finally, the simulation engine simulates a target SoS according to the marked operations.

D. Kill Checker

After executing the test cases on the original SoS policy and the mutated policies, the effectiveness of the test cases should be evaluated. In traditional mutation testing, a mutant is said to be killed if the result of the mutant is different from the result of the original program. A test case is more effective if it kills more mutants than other test cases. However, because of non-determinism, an SoS may produce different results even when executing the same test case on the same policy. Thus, we should execute the test cases several times to identify the mutants. Further, the number of needed executions and the meaning of the difference between a mutant and its original program must be defined based on the domain knowledge of the tester.

After collecting the results of repeated executions, statistical methods are applied to compare multiple results among mutants and the original program. Following the guideline for statistical analysis provided by Arcuri and Briand [24], the Mann-Whitney U test is used to compare the distribution.

V. EXPERIMENTAL DESIGN

A. Example Scenario

We extend the scenario from the previous work [25] and use as an example scenario for simulation in SUMO. The transportation SoS has been studied in the SoS research field, because it can satisfy the characteristics of an SoS. Thus, our example scenario is to control the traffic and to speed up the flow of traffic in Daejeon, South Korea. Using the graphical editor in SUMO, the target city is manually modeled as the map shown in Figure 3. The dotted circles are congested intersections.

We design 14 SoS policies to deal with traffic jams and special traffic situations like emergencies involving ambulances and few traffic. Detailed traffic jam conditions are defined as the factor of SoS policies. Each CS is comprised of

TABLE II
MUTATION OPERATORS FOR SYSTEM OF SYSTEMS POLICY

Operators	Description
Attribute Replacement (AR)	Replace an attribute of a node in a policy. For example, a target of an operation can be replaced.
Variable Replacement (VR)	Replace a variable of a node in a policy. For example, a variable of a formula to determine whether a factor is satisfied can be replaced.
Variable Deletion (VD)	Delete a variable of a node in a policy. Target is same as VR operator.
Numeric Constant Replacement (NCR)	Replace a numeric constant if a node in a policy has numeric value.
Relational Operator Replacement (ROR)	Replace a relational operator (>, <, =, $\geq$, $\leq$) with another relational operator.
Arithmetic Operator Replacement (AOR)	Replace an arithmetic operator ($+, -, \times, \div$) with another arithmetic operator.
Function Replacement (FR)	Replace a function with another function. For example, a CS function of can be replaced with another function.
Node Value Exchange (NVE)	If a policy has nodes that have same type of value, exchange values with each other.

Fig. 3. Targeted area in Daejeon. The dotted circles are congested intersections.

monitoring cameras as sensors and traffic lights as actuators in each direction, and located at each intersection. Under normal situations with no traffic jam or emergency, each traffic light of the CS in each direction changes green in a clockwise manner according to the changing cycle. However, according to the information from the monitoring cameras, if it is judged that there is a traffic jam or an emergency, the traffic light stops the cycle and follows the operation of the corresponding SoS policy. A traffic jam is determined by the number of vehicles in each direction and the sustained time of the traffic jam. Vehicle instances are stochastically generated, and detailed properties of vehicles are assigned to each instance. A route is given to an instance, and so the vehicle instance goes along its own route. In short, an instance is generated by assigning a vehicle type and a route.

B. SoS Policy for Traffic Control

Before designing an SoS policy, we define what should be included in a factor and an operation. Because the format of input files in SUMO is XML, we make the structure of the SoS policy in XML format.

We divide the factors into three parts of vehicle, location, and time. The category "vehicle" represents which type of vehicle we will monitor, for example, an ambulance or all types of vehicles. The location includes the locations we will monitor as targets, and, in order to monitor whether the factor is satisfied, the CS in charge of that location. When monitoring CSs, we observe the Number of Vehicles (NV) and the Average Waiting time (AW). These values are frequently used in traffic signal controlling algorithms [26], [27]. The satisfying state can be specified as NV(edge1)+NV(edge2)$\geq$ 30. Time is how long the factor should be satisfied to trigger the operation.

An operation also has three elements of location, time, and light. Location specifies the place where we will exert control; it could be edges, all locations, or part of the route that the vehicle instance has. Depending on the location, operating CSs are determined. Time is how long the operation is sustained and is represented by a number or a formula. Light means which light the traffic sign changes to, so the possible types of light are yellow, green, and red.

Next, we illustrate an SoS policy for an intersection with the example shown in Figure 4. Each policy has its ID and priority; the policy that has the lower priority is preferentially triggered. In the policy, factor is satisfied if the number of vehicles at edges gneE196 and gneE198 is greater than 30 and if this state continues for five ticks. The formula node contains an operator and the left and right sides of the formula, so it is NV(gneE196)+NV(gneE198)>30. Since the target of vehicles is all, any vehicles on gneE196 and gneE198 will be counted. If the factor is satisfied, traffic lights of CS on gneE196 and gneE198 will change to green for the ticks same as the total number of the vehicles on edges gneE196 and gneE198.

An SoS policy for an intersection is designed in such a way that if a specific edge is jammed, the traffic light from a jammed edge to any direction changes to green. Each direction of the intersection has its own policy, and so a three-way intersection has at least three policies. Furthermore, we have two more SoS policies that are not related to traffic jams. The first one is a policy for ambulances in order to quickly move ambulances along by changing the traffic lights for ambulances to green. The second is a policy for free passage when there are only a few vehicles on the whole map. This policy models a situation in which the traffic light is a blinking yellow light under low traffic situations, allowing vehicles to freely travel regardless of the signal. These two policies are applied throughout the whole map. Finally, the number of SoS policies in the set is 14; in detail, there are 12 policies for traffic jams and 2 policies for low traffic and ambulance situations.

After the simulation, we obtain results for the Total Sim-

```xml
<policySet>
<policy id="emergency_by_traffic_jam" priority="10">
  <factor>
        <vehicle target="all"/>
        <location target="edges">
          <formula side="left">
          NV(gneE196)+NV(gneE198)</formula>
          <formula side="operator">G</formula>
          <formula side="right">30</formula> </location>
        <time>5</time> </factor>

  <operation>
        <location target="edges">
          <edge>gneE196</edge>
          <edge>gneE198</edge></location>
        <time>NV(gneE196)+NV(gneE198)</time>
        <light>g</light> </operation>
</policy>
</policySet>
```

Fig. 4. Example of the SoS policy for a traffic jam. SoS policy is divided into a factor and an operation. The factor specifies a vehicle as a monitoring target, location, time as a monitoring CS, and a satisfying state. The operation describes the location as an operating CS, and the time and light as a detailed description of function.

ulation Time (TST) and Average Travel Time (ATT). TST indicates the overall time spent for the last vehicles to arrive after the first vehicle has left; ATT is the average time spent for each vehicle to arrive at its destination after leaving. The effectiveness of the SoS policy is assessed based on how short TST and ATT are.

C. Mutant Generation

Mutants for SoS policy testing can be automatically generated using the 8 mutation operators shown in Table II.In order to describe how to apply mutation operators, Figure 5 presents an example of applying an FR mutation operator, which replaces one function with another function. The function of traffic lights to change to green is modified to change to red. The original policy is planned to speed up the traffic flow by changing traffic lights to green under traffic jam situations; however, the mutant policy will restrict the flow by changing the light to red in spite of congestion.

```xml
<operation> ...
  <light>r</light>
  <!-- <light>g</light> -->...
</operation>
```

Fig. 5. An example of a mutant. FR operator modifies the function to change red light instead of green light.

In experiments, the eight mutation operators are applied in set of 14 SoS policies whenever possible. As a result, we generate 523 mutants.

D. Simulation

We use SUMO as the simulation engine because SUMO is capable of mimicking realistic traffic situations. However, the basic SUMO system cannot simulate dynamic changes according to operations triggered by factors during the simulation. To tackle this limitation, using SUMO APIs, we additionally implement a simulation engine wrapper to provide dynamic intervention.

E. Test Case Evaluation

To test the SoS policies, test cases can be a set of several inputs such as a traffic light algorithm, data on the types of vehicles, or generation rate of routes. A preliminary study on the composition of one test case showed that vehicle types and their routes are the main items that affect the traffic situation. We randomly generate 10 test suites considering the diversity of vehicle types and routes.

Considering the randomness of simulation, the more executions imply the more statistically sound results. However, a small number of executions is preferred in practice because of budgetary limits. In order to identify that each mutant in M is killed, each O and M are executed 10 times and there are 10 results of TST and ATT. We define the kill criterion in terms of *timeout* of TST and *distribution* of ATT. The timeout criterion distinguishes killed mutants if the TST of a mutant is over twice the average TST in O once. For example, if O takes 300 ticks on average and M takes over 600 ticks once, the execution of M is immediately terminated and M is determined as killed. Since the execution of a mutant in SoS takes a long time, the timeout criterion is crucial. If the simulations of M are not terminated by the timeout criterion, the distribution criterion is considered. Because of non-determinism, we have multiple results of ATT and must compare them at the same time between O and M. The Mann-Whitney U Test is conducted in order to compare the ATT distributions of O and M. The null hypothesis is that the distribution of M is equal to the distribution of O. If not, we say the distributions are not equal, which means that M is killed. Since 0.01 is used as the p-value, M is determined as killed if its p-value is less than 0.01.

VI. Experimental Results

A. Research Questions

The followings are the questions that we want to answer:

- RQ1: Are mutation scores correlated with fault detection ratios?
- RQ2: How long does it take to execute the mutants?
- RQ3: How much will the mutation score change if we use mutant sampling?

To address the research questions, we use Intel(R) Core i7-6700(3.40GHz) CPU, 32.0GB memory, Windows 10 Pro version of OS, and 0.25.0 version of SUMO.

B. RQ1: Correlation with Real Fault Detection

For real faults, a mutation testing expert injects 20 faults in the SoS policy. Two faults out of 20 faults are not used because they are not complied, and thus we conduct an experiment with 18 real faults. We abbreviate the SoS policy with real faults as RF01,$\cdots$,RF20, and exclude RF02 and RF04. The number of test suites is 10; these are represented as TS01,$\cdots$,TS10. The original policy O, mutants M, and

real faults RF01,···,RF20 are executed 10 times for non-determinism in our experiment. We use the kill criterion mentioned in Section V-E to determine killed mutants. The evaluation is done with the following process:

1) Execute the original policy and 523 mutant policies on all TSs 10 times. Calculate mutation scores of TSs.
2) Execute RFs on each test suite 10 times. Then observe whether a fault is detected or not.
3) Analyze the correlation between the number of detected real faults and the mutation score for each test suite.

Figure 6 shows the results. To measure correlation coefficients between the RF detection ratio and the mutation score, we calculate Pearson's linear correlation coefficient and Spearman's rank correlation coefficient. Pearson correlation is 0.951 with p-value 2.38e-05, and Spearman correlation is 0.896 with p-value 4.54e-05. This confirms that the RF detection ratio is very closely correlated with mutation score. It also implies that the proposed mutation operators listed in Table II successfully mimic the real faults made by the competent expert. As a result, it is possible to select effective test suites for SoS policies using the proposed mutation analysis approach.

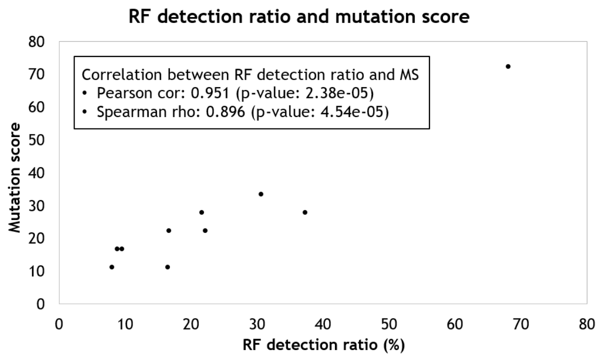

Fig. 6. Correlation between real fault detection ratio and mutation score for all test suites. There is a strong correlation between the number of detected real faults and the mutation score. A test suite with a high mutation score can detect more real faults.

C. RQ2: Execution Time of Mutants

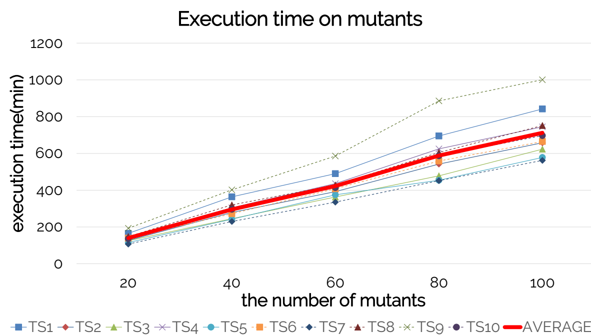

Fig. 7. Execution time of all mutants. Execution time increases linearly according to the number of mutants.

Since the time for execution is the largest part of the process of mutation testing, we decided to measure the execution time. In the experiment, we randomly extract 20, 40, 60, 80, and 100 mutants from the mutant pool, and measure how long it takes to execute a specified number of mutants in each test suite. Table 7 shows the execution time in minutes; the thickest line is the average of all test suites.

From the graphs of execution time for each test suite, time complexity according to the number of mutants is deemed to be O(n). In other words, the execution time will increase linearly depending on the number of mutants. It takes about 714 minutes to execute 100 mutants on average. However, the number of mutants can be very large, even with a small number of policies. For example, we use 14 policies in our case study, and the number of mutants is nearly 500. This means that it will take at least about 60 hours for the test suite. In Section VI-D, we seek possible solutions to reduce the execution time of mutants and to make the overall process more practical.

D. RQ3: Cost Reduction through Mutant Sampling

As discussed in Section VI-C, the cost of SoS policy testing is one of the barriers to the application of mutation analysis. In the area of traditional mutation testing, one solution to reduce the cost is mutant sampling. In mutant sampling, a subset of mutants from the entire mutant set is randomly selected and executed. One study has experimentally showed that by randomly selecting 10 percent of mutants, the mutation score is lowered by only 16 percent compared to the mutation score of the whole set. We apply this technique and compare the mutation score between sampled mutants and comprehensive mutants. In the sampling process, we randomly select mutants; the mutation score is measured in increments of 10%. This means that the mutation score of 100% is the mutation score of the whole set of mutants. Figure 8 shows the difference between the mutation score for the total mutants and the score for the mutants sampled in units of 10%.

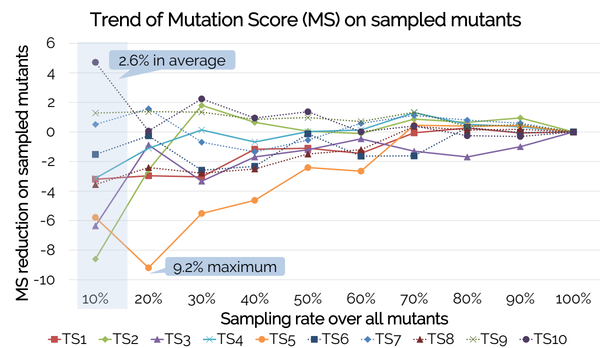

Fig. 8. Difference of mutation scores of sampled mutants and all mutants using mutation sampling. There is no significant difference between the mutation scores for sampled mutants and all mutants.

The rightmost points are the mutation scores of all mutants for each test suite; these points indicate that the sampling rate is 100%. This result shows that the difference between the mutation scores for the sampled mutants and for all mutants is less than 9.2%. The average difference when using 10%

of the whole set of mutants is 2.6%. This means that the 10% sampling rate, which is considered reasonable for general mutation testing, is appropriate even for mutation analysis for SoS policies, despite the new kill criterion. Thus, the sampling rate can be set to reduce costs, depending on the budget and the accuracy of results desired by the tester. Higher sampling rates will bring about results much closer to the mutation scores of comprehensive mutants.

VII. Conclusion

This paper suggests a mutation analysis approach for SoS policy testing. We define 8 mutation operators for SoS policies and a kill criterion to deal with the non-determinism of the SoS. In a case study, a traffic congestion control SoS is simulated by SUMO.

For the experiment, we try to find answers in three areas: correlation between mutation score real faults detection ratio, time complexity, and the possibility of cost reduction through the mutant sampling. First, there is a strong correlation between mutation score and real fault detection ratio. It means that test suites with a higher mutation score can detect more real faults. Second, the time taken to execute mutants increases linearly according to the number of mutants: 714 minutes are needed to execute 100 mutants on average. Third, if the sampling rate is 10%, which is the general sampling rate in mutation testing, the score difference is 2.6% on average. This means that testers can reduce the cost by sampling about 10% of mutants, in spite of the different kill criterion.

Our future work will be to apply existing test data generation techniques to SoS policy testing. We will perform experiments with other simulators to test SoS policy in other SoS domains. Through experiments with SoS policies in other domains, the mutation operators and the kill criterion that are statistically defined in this paper can be further evaluated.

Acknowledgment

This research was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No.R0126-16-1101, (SW Star Lab) Software R&D for Model-based Analysis and Verification of Higher-order Large Complex System. We also would like to thank anonymous reviewers for their valuable comments.

References

[1] M. W. Maier, "Architecting principles for systems-of-systems," *INCOSE International Symposium*, vol. 6, no. 1, pp. 565–573, 1996.

[2] M. Hall-May and T. Kelly, "Defining and decomposing safety policy for systems of systems," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2005, pp. 37–51.

[3] D. DeLaurentis, R. K. Callaway *et al.*, "A system-of-systems perspective for public policy decisions," *Review of Policy Research*, vol. 21, no. 6, pp. 829–837, 2004.

[4] D. B. Agusdinata and D. DeLaurentis, "Specification of system-of-systems for policymaking in the energy sector," *Integrated Assessment*, vol. 8, no. 2, 2008.

[5] B. Agusdinata, *Exploratory modeling and analysis: a promising method to deal with deep uncertainty*. TU Delft, Delft University of Technology, 2008.

[6] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[7] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 654–665.

[8] D. DeLaurentis, "Understanding transportation as a system-of-systems design problem," in *43rd AIAA Aerospace Sciences Meeting and Exhibit*, vol. 1. Reno, NV New York, NY, 2005.

[9] M. Mansouri, A. Gorod, T. H. Wakeman, and B. Sauser, "Maritime transportation system of systems management framework: A system of systems engineering approach," *International Journal of Ocean Systems Management*, vol. 1, no. 2, pp. 200–226, 2009.

[10] C. B. Nielsen, P. G. Larsen, J. Fitzgerald, J. Woodcock, and J. Peleska, "Systems of systems engineering: basic concepts, model-based techniques, and research directions," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 18, 2015.

[11] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker, "Recent development and applications of SUMO - Simulation of Urban MObility," *International Journal On Advances in Systems and Measurements*, vol. 5, no. 3&4, pp. 128–138, December 2012.

[12] W. E. Walker, "Policy analysis: a systematic approach to supporting policymaking in the public sector," *Journal of Multicriteria Decision Analysis*, vol. 9, no. 1-3, p. 11, 2000.

[13] D. B. Agusdinata and L. Dittmar, "Adaptive policy design to reduce carbon emissions: a system-of-systems perspective," *IEEE Systems Journal*, vol. 3, no. 4, pp. 509–519, 2009.

[14] A. Mostafavi, D. Abraham, and D. DeLaurentis, "Ex-ante policy analysis in civil infrastructure systems," *Journal of Computing in Civil Engineering*, vol. 28, no. 5, p. A4014006, 2013.

[15] A. Mostafavi, "Integrated policy simulation in complex system-of-systems," in *Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World*. IEEE Press, 2013, pp. 4004–4005.

[16] S. Luna, A. Lopes, H. Y. S. Tao, F. Zapata, and R. Pineda, "Integration, verification, validation, test, and evaluation (ivvt&e) framework for system of systems (sos)," *Procedia Computer Science*, vol. 20, pp. 298–305, 2013.

[17] Q. Liang and S. H. Rubin, "Randomization for testing systems of systems," in *Information Reuse & Integration, 2009. IRI'09. IEEE International Conference on*. IEEE, 2009, pp. 110–114.

[18] O. AT&L, "Systems engineering guide for systems of systems," *Washington, DC: Pentagon*, 2008.

[19] R. Lipton, "Fault diagnosis of computer programs," *Student Report, Carnegie Mellon University*, 1971.

[20] R. J. Lipton, R. A. DeMillo, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE computer*, vol. 11, no. 4, pp. 34–41, 1978.

[21] R. Geist, A. J. Offutt, and F. C. Harris, "Estimation and enhancement of real-time software reliability through mutation analysis," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 550–558, 1992.

[22] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.

[23] N. T. Binh *et al.*, "Mutation operators for simulink models," in *Knowledge and Systems Engineering (KSE), 2012 Fourth International Conference on*. IEEE, 2012, pp. 54–59.

[24] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[25] W. Yun, J. Song, E. Jee, and D.-H. Bae, "A suitability analysis of sumo for modeling system of systems in traffic domain example," in *2016 Korea Computer Congress*. Korean Computer Congress, 2016, pp. 631–633.

[26] D. An and G. Cho, "Adaptive traffic light control system in vanet environment," *2012 Korea Computer Congress*, vol. 39, no. 1D, pp. 343–345, 2012.

[27] H.-S. Lee, W.-H. Han, and W.-K. Choi, "Improved crossroad signal system using etm," *2008 Korea Entertainment Industry Association*, vol. 2, no. 2, pp. 517–523, 2008.