Automated Derivation of Test Requirements for Systems of Systems

Jhonatan Azevedo Gonçalves
Universidade Federal Fluminense
Niterói, Brazil
jhonatan_goncalves@id.uff.br

Francesca Lonetti
ISTI–CNR
Pisa, Italy
francesca.lonetti@isti.cnr.it

Vânia de Oliveira Neves
Universidade Federal Fluminense
Niterói, Brazil
vania@ic.uff.br

***Abstract*—Testing of Systems of Systems (SoS) is challenging and improving its cost-effectiveness is a relevant research topic. In this paper, we propose TESoS (Test Engine for Systems of Systems), a systematic approach that selects from SoS models, defined in mKAOS language, the functionalities to be tested and then automatically derives a set of test requirements. TESoS allows to classify test requirements according to unit, integration, and system testing levels. Moreover, it helps test planning by providing the tester with automated facilities for supporting the unit testing of constituent systems and computing the percentage of test requirements that are satisfied with a given test suite. We illustrate the TESoS application on an SoS case study in the educational domain.**

***Index Terms*—Systems of Systems, Automated Testing, Test Requirements, mKAOS**

I. Introduction

Systems of Systems (SoSs) are today developed in many application domains for dealing with the increasing complexity and the dynamic evolution of modern software systems. SoSs are composed of several systems called Constituent Systems (CSs). CSs have their own architecture and capabilities and are developed to meet a particular purpose. Integrating specific CSs capabilities allows to fulfill an overall SoS mission that goes beyond each CS's individual capability.

The dynamic and evolving behavior of SoSs pose many challenges to their test and evaluation [1]. Operational and managerial independence, evolutionary development, emergent behaviors and decentralized nature are the main characteristics of SoSs that pose too many testing issues at the different testing levels. Testing activities of traditional systems are structured into three main phases: unit, integration, and system test. The first is responsible for ensuring the functionality of the system's smallest unit. The second guarantees that the integration between different units, or classes of units, works as expected. And the last one usually acts on top of non-functional requirements, ensuring the operation of the integrated software as a whole [2]. Neves et al. [3] provide a summary of SoS features and related testing issues at the different levels of software testing, i.e., unit, integration, and system level, also suggesting existing testing strategies that could be adapted for SoS testing.

Among the SoS testing issues, there is a lack of SoS testing requirements. Since an SoS is dynamically assembled to accomplish a specific mission, it has no requirements per se. In many cases, requirements are specified for constituent systems but not for the SoS [1]. However, not all the functionalities of CSs are used to fulfill the SoS mission. This makes it difficult to define the scope of what should be tested in the SoS and poses the need for a systematic approach to select the functionalities of CSs to be tested and the derivation of a set of test requirements for the SoS [3]. Our proposal goes in this direction and aims to provide an automated approach for test requirements derivation starting from the SoS models.

Different languages and paradigms have been adopted for defining SoSs [4]. Such languages for SoS design are complex and their knowledge is often not included in the tester's background. Among the SoS modeling languages, mKAOS [5] is a commonly adopted mission-based language for designing SoS architecture, allowing for the definition of the missions, the capabilities of the constituent systems required to achieve such missions, as well as the interactions among them [6], [7]. However, the literature does not report specific automated engines or approaches for deriving test requirements from complex mission-based models, specified into mKAOS.

In this paper we provide an approach for deriving SoS test requirements from mKAOS models. We first define a simple strategy for identifying unit, integration, and system test requirements of an SoS modeled into mKAOS and then we propose TESoS (Test Engine for Systems of Systems) that implements this strategy for automatically deriving a list of test requirements. TESoS is able to process three types of mKAOS models that are *operational capability models*, *emergent behavior models*, and *mission models*. Through the coverage of these models, it derives a set of test requirements classified according to each level of software testing: unit, integration, and system. The proposed approach represents a valid support for the SoS testing activities, since it enables the tester: i) to automatically identify the SoS test requirements hiding the complexity of the SoS models defined during the SoS design; ii) to manage a set of API (Application Programming Interface) links that provide support for automated unit testing of CSs; iii) to check what test requirements have been satisfied. TESoS includes an open-source tool that is available in the GitHub TESoS repository[1].

[1]https://github.com/educationalsos/tesos

As an application example, we applied our approach to "International Master", i.e., an SoS in the distance learning and educational domain belonging to *EDUFYSoS* [8].

The remaining of the paper is structured as follows: in the next section, we present some background and more closely related work. In Section III, we present an overview of TESoS while in Section IV, we describe the architecture of the TESoS tool and its main components. In Section V we demonstrate the TESoS application to a case study. Finally, in Section VI we draw conclusions, also discussing limitations and future research directions.

II. Background and Related Work

SoSs represent a new paradigm of complex systems that emerge from the combination of pre-existing independent systems (constituent systems) with the aim of accomplishing a global collaborative mission beyond the functionalities of the individual constituent systems [9]. SoSs are nowadays developed in many domains, including transportation, security, healthcare, home automation, military or emergency systems. According to the categorization of SoSs from the US Department of Defense [10], four different SoS architectures are known: i) directed SoS, in which a central system is responsible for the fulfillment of the global purpose; ii) acknowledged SoS, in which constituent systems retain self-control but are coordinated by a central authority; iii) collaborative SoS, in which there is mutual agreement on performing certain jobs and reaching a purpose; iv) virtual SoS, in which there is no central management authority and an explicit shared goal among the CSs, which might not be aware they are working for a global purpose.

Several architecture description languages, such as SosADL [11] or SysML profiles [12] have been specifically conceived to support automated SoS modeling and analysis. mKAOS is a language for SoS mission modeling [5] described as a specialization of KAOS (Keep All Objectives Satisfied) i.e., a goal-oriented requirements engineering method [13]. mKAOS allows to organize the mission related information in a set of complementary models that provide a description of the mission information independently from the implementation details. There are six different kinds of models in mKAOS [5], [14]: i) *mission models* where the leaf nodes represent individual missions associated with the CS, and non-leaf nodes represent global missions. Refinement links establish a refinement relationship among missions, enabling the refining of a specific mission into other sub-missions; ii) *responsibility models* representing the assignment of specific responsibilities tied to individual missions to constituent systems. They can also define software agents and their responsibilities with respect to the SoS goals; iii) *object models* representing entities and events that characterize the system; iv) *operational capability models* defining a set of specific operations that each constituent system must implement to contribute to the mission achievement. These operations can trigger events while they can handle entities as inputs and outputs. Each constituent system must have its own operational capability model; v) *communicational capability models* that represent the connectivity and the interactions among constituent systems. There is a single communicational capability model for the whole system; vi) *emergent behavior models* that define emergent behaviors of SoSs achieved through the cooperation among different constituent systems. The emergent behavior model represents a composition of communicational capabilities. mKAOS Studio [14] is an open-source tool, released as an Eclipse plugin, for modeling missions in SoS by using the mKAOS language and building XML files based on these models. In the proposed approach, TESoS targets *operational capability models*, *emergent behavior models*, and *mission models* for the automatic derivation of test requirements as we will explain in Section III.

One of the challenging core activities of SoS Systems Engineering is to translate the SoS capabilities and goals into SoS requirements and validate them [15]. The goal of our approach goes in this direction. TESoS allows deriving a list of SoS test requirements from the SoS models, supporting the tester in the automation of the testing activities. Our approach can be applied to the four different SoS architectures described before, since SoS mKAOS models have been designed.

The automated identification of test requirements has been addressed in the literature and several test strategies based on coverage criteria have been proposed for deriving test requirements from UML models. For instance, Nebut et al. [16] propose an approach for deriving test objectives as a set of instantiated use cases covering an enhanced UML use cases model of a product in the context of software product lines. Briand et al. [17] propose a technique for identifying a set of paths from a UML statechart and deriving a set of test requirements covering these paths. The idea of deriving a set of test requirements covering all the main entities of the model is similar in our proposal. Differently from previous approaches, we target SoS testing and mKAOS models aiming to generate a set of test requirements covering the main entities of mKAOS models, i.e. all the operational capabilities, the emergent behaviors and communication capabilities as well as mission and sub-missions of mKAOS models.

Concerning testing of SoS, beside many challenges have been identified at different testing levels, not much research on SoS testing yet exists. The authors of [18] make a parallel between the traditional testing levels and the SoS testing levels trying to identify the significant challenges of SoS testing at all levels. Moreover, the authors of [3] discuss how existing test techniques can be adapted to deal with specific features of SoS. Testing SoS at the unit level deals with the testing of each individual constituent system independently from the SoS in which it is involved, then traditional testing strategies can be applied [18]. At SoS integration testing level, Luna et al. [19] propose combinatorial testing strategies whereas Liang et al. [20] adopt a randomization approach to design integration test cases. Other approaches address system testing of SoSs. For instance, Zapata et al. [21] use a control flow graph for modeling SoS and apply a path testing technique for deriving

test cases while the authors of [22] propose an adaptive testing framework for planning tests according to critical test events. More recently, the work in [6] provides an approach for modeling and testing of SoSs taking into account the variable functionalities of CSs and their costs, leveraging software product lines based solutions. The approach is also able to support test planning with the derivation of test objectives and test scenarios. Differently from previous solutions aiming to derive test cases, the goal of our proposal is to automatically derive test requirements from mKAOS models.

III. TESoS Overview

SoSs can be composed of several constituent systems, which, in turn, offer several functionalities. Therefore, due to the dynamic and complex nature of SoSs, the number of test requirements can be huge, which makes manual testing activity impractical. For this, we propose the TESoS approach, which allows us to identify the functionalities of each constituent system as well as of the overall SoS and then derive a set of test requirements. Also, since testers may not have enough SoS knowledge or be unfamiliar with SoS design models, TESoS provides a useful support for identifying test requirements in an automated way. In the following, we show the simple strategy adopted in TESoS for identifying test requirements (Section III-A) and then the main steps of the TESoS approach (in Section III-B).

A. Identifying SoS test requirements from mKAOS models

TESoS derives test requirements for each level of SoS testing: unit, integration, and system testing. To this end, it considers the mKAOS models generated during the SoS design phase, as explained below.

Previous studies [3] have identified that at the unit testing level, each individual CS of the SoS should be tested. This means that to perform the unit testing, all the operational capabilities of a CS need to be tested. The mKAOS operational capability model defines a set of specific operations that each CS must implement to contribute to the fulfillment of the SoS mission. In TESoS we derive unit test requirements for a CS in order to cover all the operational capabilities of the mKAOS operational capability model defined for that CS.

At the integration testing level, all the possible interactions among all the CS which dynamically can join or leave the SoS need to be tested. In SoSs, the interaction among CSs occurs when emergent behavior takes place. The mKAOS emergent behavior model defines all the cooperations among different CSs expressed through the composition of communication capabilities. So, in TESoS we derive integration test requirements for an SoS to cover all the emergent behaviors and the associated communication capabilities of the mKAOS emergent behavior model defined for that SoS.

Finally, the SoS system testing level corresponds to the testing of the proper behavior or mission of the whole SoS. In mKAOS, mission model represents the SoS global mission refined into more sub-missions. In TESoS we derive system test requirements for an SoS to cover the mission and submissions of the mKAOS mission model defined for that SoS.

B. TESoS main steps

In order to derive the test requirements, it is necessary that an SoS Engineer, who has proper knowledge about the SoS and its CSs, designs the mission and the emergent behavior model, as well as the operational capabilities model for each constituent system, as illustrated in the first step of Figure 1. The SoS design can be done using the mKAOS Studio tool. In this tool, the mission model and the emergent behavior model are saved in a single integrated XML file; the models of operational capabilities of the CSs are stored in separate XML files. TESoS can read these XML files to generate the test requirements, as illustrated in step two of Figure 1. It is noteworthy, however, that the inputs for generating the test requirements in TESoS are the XML files that could have been developed by any other tool or even created manually.

After designing the SoS, the next step is to upload the XML files containing the models into TESoS (step two of Figure 1). In this step, TESoS asks the SoS engineer for the URI and API links of the CSs services used in the SoS. This information is needed to help the tester implementing the test cases. Considering that each constituent system can have several functionalities and even with similar characteristics, at this stage, the SoS engineer defines which specific functionality s(he) considered during the design. Thus, we expect that the test case that will be designed later by the tester would include at least one call to these APIs.

Once the SoS is registered and configured, TESoS identifies the test requirements and classifies them according to the testing level, presenting them on an interactive table, as described in step three of Figure 1. The tester will be able to consult the generated test requirements that may guide him/her in preparing test cases. For instance, considering an SoS unit test requirement, TESoS will display which functionality and its respective CS API needs to be exercised by the test case to meet this requirement.

Once the test cases are created, the tester can check whether or not a test requirement is satisfied, as depicted in step four of Figure 1. In the current version of TESoS, the tester performs this step manually. Then, TESoS shows the percentage of covered test requirements, i.e., the number of test requirements that have been met by the test cases with respect to the total number of test requirements, as illustrated in step five of Figure 1.

IV. TESoS Tool Architecture

The TESoS approach includes the TESoS tool for the automated generation of test requirements. This tool has a web-based architecture that follows the widely adopted Model/View/Controller (MVC) design pattern. The MVC pattern organizes the information flow into three levels: the Model, the View, and the Controller. Figure 2 displays the architecture diagram of the TESoS tool, which consists of two main independent modules: the *server* and the *client*. The

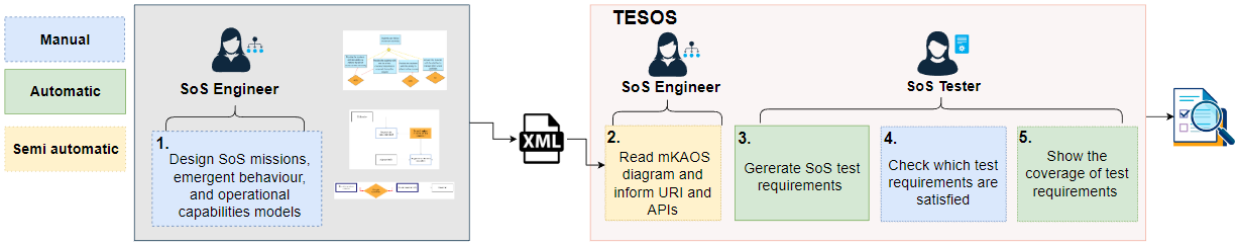

Fig. 1. TESoS Overview

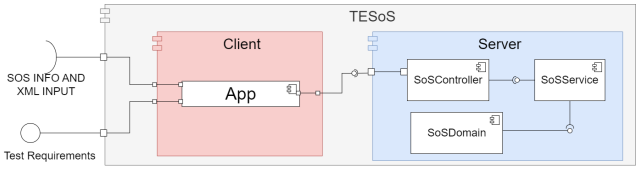

Fig. 2. TESoS tool Architecture

server module hosts the Model and Controller layers, while the *client* module contains the View layer.

The *server* module is represented by light blue color in Figure 2. It has been implemented using Spring Boot [2], and it is composed of three components: *SoS Domain*, *SoSService*, and *SoSController*. The *SoSDomain* component is formed by classes representing the system's main entities. In this component, a class is created for each entity-relationship model entity. It is implemented using the Java Persistence API (JPA) framework, representing a simple and easy solution for database management and reliable integration with Spring Boot [23].

The *SoSService* component aims to control access to the stored data in the TESoS tool. To achieve this, we used the Data Access Object (DAO) approach, and we were able to define all the necessary queries of the entities. The DAO pattern acts as an intermediary layer between the data and the business model, contributing to encapsulating the code. In our implementation, we created an abstract DAO class containing the methods for the basic persistence CRUD (Create, Retrieve, Update, Delete) operations. We then developed service classes that correspond to the entities in TESoS tool. These services classes extend the abstract DAO class and implement additional queries if needed.

The *SoSController* component is responsible for handling user requests from the client through HTTP requests. This component processes the request, calls the appropriate service functions, and returns a response containing the retrieved data.

The *client* module is represented in Figure 2 in light red color. It comprises the *App* component implemented using the Angular framework. It is in charge of allowing user interaction with TESoS and also providing data visualization. Specifically, it has two main objectives: receiving and organizing the data that are stored and processed in the server and presenting the test requirements. This module mainly provides Graphical User Interfaces (GUIs) for SoS engineers and SoS testers. The SoS engineer's GUIs consist of web forms that enable these engineers to provide the essential information about the SoS, the XML files with the models, as well as the API links of the operational capabilities of each constituent system, as explained in Section III. These web forms also allow for registering users and managing all SoSs registered in TESoS. The SoS tester can also access the list of registered SoSs and select the desired SoS to perform the test. Once the SoS is selected, a web page with the test requirements is loaded. The test requirements are organized according to unit, integration, and system testing levels. Aside from the test requirements, this GUI also provides other facilities to support the tester job, such as the API links for unit testing and a requirements check box useful for the computation of test requirements coverage by a given test suite. Since TESoS is a web based tool, aside from the parsing algorithm, the computational time of the majority of the operations is irrelevant. Even though we used a simple parsing algorithm that depends on the size of the model, the wait time of the parsing is reasonable for a web based application, even for large models.

V. APPLICATION EXAMPLE

In this section, we show the application of our approach to an SoS case study in the educational domain that is the "International Master" SoS belonging to *EDUFYSoS* [8] and available at *EDUFYSoS* repository[3]. This SoS case study refers to an international master's degree offered by two joining universities that want to provide the students with a multicultural learning experience. The constituent systems involved in the "International Master" are:

- Two administrative office systems responsible for managing information about courses and students from both universities joining the international master degree. Rosarios[4] and Sapos[5] respectively implement these systems.
- A cooperative administrative office system that allows managing the information about courses and students of the international graduate program defined across both universities. This system is implemented as a functionality in Rosarios.

[2]https://spring.io/

[3]https://github.com/edufysos/edufysos
[4]www.rosariosis.org
[5]http://gems-uff.github.io/sapos/

- The learning management system responsible for delivering online educational courses, enabling teachers to create assignments, track student progress, and report on results. This system is implemented by FullTeaching[6].
- A calendar system that allows for time management and provides the user with the ability to manage and share meetings, events, and deadlines. This system is implemented by Google Calendar[7].

The first step of the proposed approach deals with the definition of mKAOS models for the "International Master" SoS. Specifically, the SoS engineer needs to design three types of SoS diagrams: mission diagram, emergent behavior diagram, and operational capability diagrams.

Figure 3 shows the "International Master" SoS mission diagram designed into mKAOS. In the figure, the missions are represented by blue rectangles, the refinements of the missions are represented by yellow circles, while the orange diamonds represent the abstract CSs. The general mission of the "International Master" SoS is "*A student with an international master offered by two universities is able to properly manage it*". This general mission is refined into two sub-missions that are: "*The two universities are able to offer the international master*" and "*The student is able to follow his international master*". Each of them is further refined into other more specific sub-missions that represent the capabilities of the associated abstract CSs.

An extract of the emergent behavior model regarding the sub-mission "*Provide the students with the ability to manage their courses activities*" of "International Master" SoS is depicted in Figure 4. The orange rectangles represent the emergent behaviors, while the sky blue rectangles associated with each emergent behavior represent the communication capabilities, i.e., the actions performed on the CSs that needed to be tested for the achievement of the SoS mission and that are associated with the entities (white rectangles).

Figure 5 shows the operational capability diagram of RosarioSis, the concrete CS implementing both the administrative office system of one of the two universities and the cooperative administrative office of "International Master" SoS. In that diagram, blue rectangles represent a set of functionalities offered by RosarioSis for the achievement of the SoS mission. For the aim of simplicity, in this paper we only show the operational capability diagram of RosarioSis. We refer to TESoS repository [8] for the complete versions of "International Master" SoS emergent behavior model and mission model as well as the operational capability diagrams of the other CSs involved in the "International Master" SoS, i.e., FullTeaching, Google Calendar, and Sapos.

Starting from all the mKAOS models defined for the "International Master" SoS, XML files are generated and given as input to TESoS. TESoS is able to read these XML files and identify all entities belonging to the mission diagram, emergent behavior diagram, and operational capability diagrams. Then, applying the simple strategy described in Section III-A, the TESoS tool automatically derives a set of unit, integration, and system test requirements for "International Master" SoS.

As an example, Figure 6 depicts a screenshot with the unit test requirements visualized by the tester. The TESoS tool also shows the concrete CS to which each unit test requirement refers to (second column in Figure 6). For instance, the first seven test requirements visualized in the screenshot refer to RosarioSis, while the last requirements refer to Google Calendar. Moreover, the TESoS tool also shows the API link (fourth column in Figure 6) associated with each unit test requirement, i.e., a link to an API associated with the concrete CS to which the unit test refers to and that allows to perform a test action directly on the selected CS. For instance, for the first unit test requirement "*To add student*" of Figure 6, the associated API link ($rosariosis/addStudent.php$) allows to directly add an event on the RosarioSis CS. The tester will then know that (s)he must create a test case in which this API is called, passing the appropriate parameters to create a student in RosarioSis CS. Moreover, TESoS also allows the tester to check and mark which are the test requirements that have been satisfied, for instance, with an existing test suite, as depicted in the last column of the screenshot of Figure 6. In this example, four out of nine unit test requirements are checked, meaning that there are test cases that meet these requirements, and therefore, the coverage percentage calculated by TESoS is 44%. In the case of integration test requirements, TESoS shows the list of emergent behaviors and the associated communication capabilities extracted from the emergent behavior model. For instance, as depicted in Figure 7, "To provide calendar info", "To provide classes schedule", and "To add new task" are the communication capabilities associated with "To add online classes on his calendar" emergent behavior of "International Master" SoS. To satisfy these requirements, the tester must then create test cases that exercise these three communication capabilities sequentially. Finally, for the system test requirements, TESoS shows the list of missions and sub-missions of the SoS mission diagram (see Figure 8). As for unit test requirements, also for integration, and system test requirements, TESoS allows the tester to check and mark what are the test requirements that have been satisfied, as depicted in the last column of the screenshots of Figure 7 and Figure 8. We refer to TESoS repository for the complete list of unit, integration and system test requirements of the provided case study.

VI. Conclusions and Future Work

SoSs are dynamic and evolving systems that can provide several functionalities. As such, deriving test requirements can become impractical if carried out manually. In this paper, we proposed TESoS, an automated approach for identifying the functionalities to be tested in the SoS and deriving unit, integration, and system test requirements from mKAOS diagrams, described in XML format. By the application to a case study in the educational domain, we demonstrated how TESoS can

[6]https://github.com/pabloFuente/full-teaching
[7]https://developers.google.com/calendar
[8]https://github.com/educationalsos/tesos

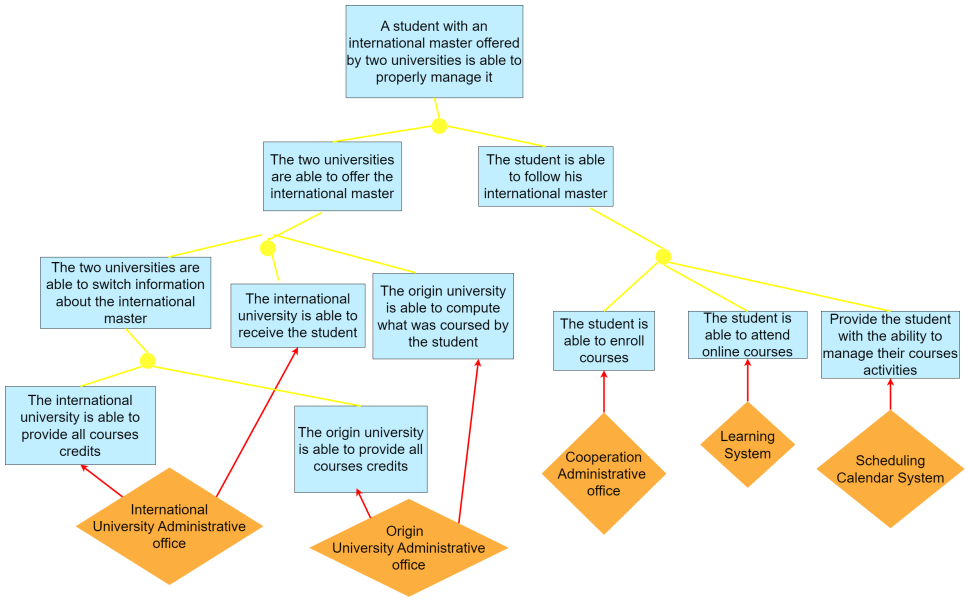

Fig. 3. "International Master" SoS mission diagram

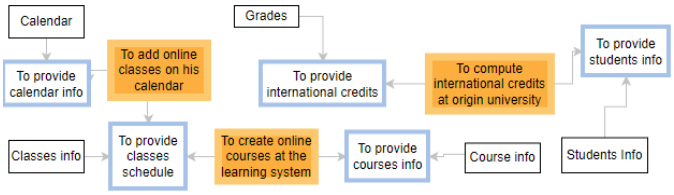

Fig. 4. "International Master" SoS emergent behavior diagram

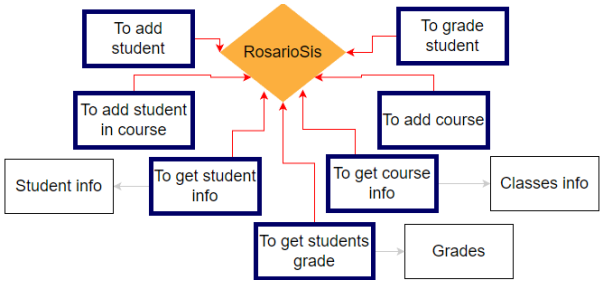

Fig. 5. RosarioSis operational capability diagram

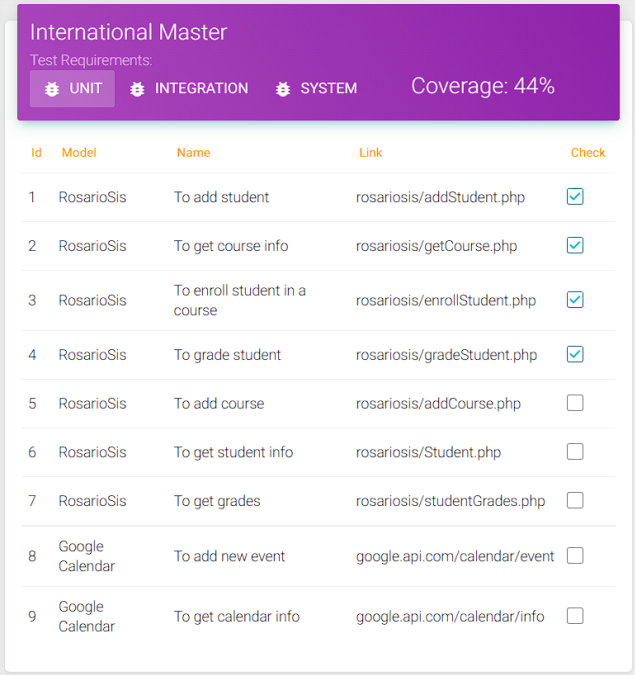

Fig. 6. Visualization of unit test requirements

guide testers in defining what can be tested in SoSs, providing useful facilities for supporting test automation.

However, TESoS has some limitations. One limitation is that the CSs that are part of the SoS must be web systems that communicate via APIs. Another limitation of TESoS is that it currently only supports mKAOS models, although the TESoS architecture is generic and independent of any formalism. However, since TESoS parses XML files, another type of SoS representation can be easily considered by transforming this representation into XML files. In the future, we plan to extend TESoS to consider other SoS representations, such as SysML [12] and SoSADL [11], to derive test requirements.

TESoS represents the first step towards automating the derivation of test cases, but in the current version of the approach, some steps require manual or semi-automatic intervention. To address this limitation, in future works we will focus on fully automating the approach. For step four, presented in Figure 1, for instance, we plan to develop an algorithm that can automatically identify which test requirements are met based on a repository of test cases. This identification would consider, for instance, if the APIs info informed by the

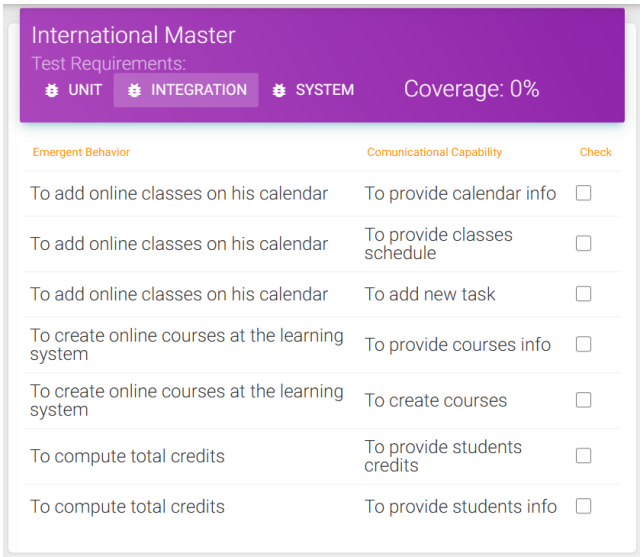

Fig. 7. Visualization of integration test requirements

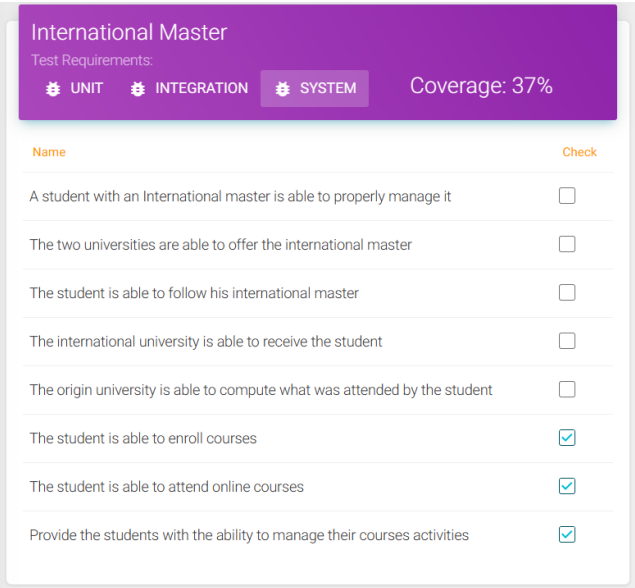

Fig. 8. Visualization of system test requirements

SoS engineer during the SoS design is called in the test cases implementation. In addition, we plan to automatically generate test scripts that satisfy the identified test requirements. Finally, we plan to evaluate the efficiency and effectiveness of TESoS on larger, real-world SoSs.

Acknowledgment

This work was partially supported by the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU, and the PIBIC PROPPI/UFF/CNPq Grant.

References

[1] J. Dahmann, J. A. Lane, G. Rebovich, and R. Lowry, "Systems of systems test and evaluation challenges," in *Proc. of SoSE*, 2010, pp. 1–6.

[2] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *FOSE*. IEEE, 2007, pp. 85–103.

[3] V. de Oliveira Neves, A. Bertolino, G. De Angelis, and L. Garcés, "Do we need new strategies for testing systems-of-systems?" in *Proc. of SESoS*, 2018, pp. 29–32.

[4] C. A. Lana, M. Guessi, P. O. Antonino, D. Rombach, and E. Y. Nakagawa, "A systematic identification of formal and semi-formal languages and techniques for software-intensive systems-of-systems requirements modeling," *IEEE systems journal*, vol. 13, no. 3, pp. 2201–2212, 2018.

[5] E. Silva, T. Batista, and F. Oquendo, "A mission-oriented approach for designing system-of-systems," in *Proc. of SoSE*, 2015, pp. 346–351.

[6] F. Lonetti, V. de Oliveira Neves, and A. Bertolino, "Designing and testing systems of systems: From variability models to test cases passing through desirability assessment," *Journal of Software: Evolution and Process*, vol. 34, no. 10, p. e2427, 2022.

[7] M. Daun, J. Brings, L. Krajinski, V. Stenkova, and T. Bandyszak, "A GRL-compliant iStar extension for collaborative cyber-physical systems ," *Requirements Engineering*, vol. 26, no. 3, pp. 325–370, 2021.

[8] A. Bertolino, G. De Angelis, F. Lonetti, V. de Oliveira Neves, and M. A. Olivero, "Edufysos: A factory of educational system of systems case studies," in *Proc. of SoSE*. IEEE, 2020, pp. 205–210.

[9] M. W. Maier, "Architecting principles for systems-of-systems," *Systems Engineering: The Journal of the International Council on Systems Engineering*, vol. 1, no. 4, pp. 267–284, 1998.

[10] J. S. Dahmann and K. J. Baldwin, "Understanding the current state of us defense systems of systems and the implications for systems engineering," in *Proc. of the 2nd Annual IEEE Systems Conference*, 2008, pp. 1–7.

[11] F. Oquendo, "Formally describing the software architecture of systems-of-systems with SosADL," in *Proc. of SoSE*. IEEE, 2016, pp. 1–6.

[12] M. Mori, A. Ceccarelli, P. Lollini, B. Frömel, F. Brancati, and A. Bondavalli, "Systems-of-systems modeling using a comprehensive viewpoint-based sysml profile," *Journal of Software: Evolution and Process*, vol. 30, no. 3, p. e1878, 2018.

[13] A. Van Lamsweerde and E. Letier, "From object orientation to goal orientation: A paradigm shift for requirements engineering," in *Radical Innovations of Software and Systems Engineering in the Future*. Springer, 2004, pp. 325–340.

[14] E. Silva, T. Batista, and E. Cavalcante, "A mission-oriented tool for system-of-systems modeling," in *Proc. of SESoS*, 2015, pp. 31–36.

[15] J. S. Dahmann, "Systems of systems characterization and types," *Systems of Systems Engineering for NATO Defence Applications (STO-EN-SCI-276)*, pp. 1–14, 2015.

[16] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, "Automatic test generation: A use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140–155, 2006.

[17] L. C. Briand, Y. Labiche, and J. Cui, "Automated support for deriving test requirements from UML statecharts," *Software & Systems Modeling*, vol. 4, pp. 399–423, 2005.

[18] A. Bertolino, F. Lonetti, and V. de Oliveira Neves, "Standing on the shoulders of software product line research for testing systems of systems," in *Proc. of ISSREW*. IEEE, 2020, pp. 209–214.

[19] 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.

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

[21] F. Zapata, A. Akundi, R. Pineda, and E. Smith, "Basis path analysis for testing complex system of systems," *Procedia Computer Science*, vol. 20, pp. 256–261, 2013.

[22] J. T. Hess and R. Valerdi, "Test and evaluation of a SoS using a prescriptive and adaptive testing framework," in *Proc. of SoSE*. IEEE, 2010, pp. 1–6.

[23] O. Gierke *et al.*, "Spring Data JPA-Reference Documentation," https://docs.spring.io/spring-data/jpa/docs/current/reference/html/, 2023.