Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Designing and Testing Systems of Systems: From Variability Models to Test Cases passing through Desirability Assessment[†]

Francesca Lonetti*[1] | Vânia de Oliveira Neves[2] | Antonia Bertolino[1]

[1]Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR, Pisa, Italy
[2]Universidade Federal Fluminense, Niterói, Brazil

Correspondence
*Francesca Lonetti Email: francesca.lonetti@isti.cnr.it

Summary

In the early stages of a System of Systems (SoS) conception, several constituent systems could be available that provide similar functionalities. An SoS design methodology should provide adequate means to model variability in order to support the opportunistic selection of the most desirable SoS configuration. We propose the VANTESS approach that: i) supports SoS modelling taking into account the variation points implied by the considered constituent systems; ii) includes a heuristics to weight benefits and costs of potential architectural choices (called as SoS variants) for the selection of the constituent systems; and finally iii) also helps test planning for the selected SoS variant by deriving a simulation model on which test objectives and scenarios can be devised. We illustrate an application example of VANTESS to the "Educational" SoS and discuss its pros and cons within a focus group.

KEYWORDS:
System of Systems, Design, Software Product Line, Testing, Variability Model, Test Case Generation

1 | INTRODUCTION

A system of systems (SoS) is obtained by connecting a set of constituent systems (CSs) so that altogether they can achieve a global goal or mission that is beyond their individual capabilities. Depending on the existence of a central management and on the degree of awareness and commitment of the CSs to cooperate, four different SoS architectures, progressively more loosely connected, are distinguished, namely Directed, Acknowledged, Collaborative, and Virtual[1].

SoSs are usually dynamic, distributed and complex systems, whose design, maintenance and validation pose several research challenges[2]. In fact, by definition the CSs retain operational and managerial independence, and may even evolve in unforeseen manner, giving rise to unpredictable SoS behaviors. Unfortunately, as early observed by DeLaurentis[3], "traditional" design methods and tools, even if conceived for large distributed (but monolithic) systems, may not be applicable to SoSs. The problem is that generally the CSs were not originally designed to be later merged into the SoS. Indeed, many crucial services to which we are now acquainted in modern society and smart cities are actually provided by SoSs that were formed and evolved "into place"[3].

An important aspect in SoSs is variability, concerning both time and space[4]: within an SoS context, variability in time, or evolution, for an SoS refers to the occurrence of different versions of a CS at different times, whereas variability in space refers to the availability at the same time of different artifacts for a same CS. The scope of this paper is SoS variability in space.

[†]This is the pre-peer reviewed version of the following article: [Lonetti F, de Oliveira Neves V, Bertolino A. Designing and testing systems of systems: From variability models to test cases passing through desirability assessment. J Softw Evol Proc. 2022;e2427. doi:10.1002/smr.2427], which has been published in final form at http://doi.org/10.1002/smr.2427. This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Use of Self-Archived Versions."

Different languages and paradigms have been adopted for the modeling of SoS requirements [5], and several approaches have been proposed for the engineering of SoS [2]. However, the problem of managing SoS variability has not been adequately addressed. We have found few approaches [6,7] supporting the modeling of the SoS architecture at abstract level that can tackle the natural environment evolution, but many open challenges remain [8,9] . In particular, among existing approaches we did not find any one allowing for the handling of space variability already at SoS early design. In this respect, we notice that SoS design is inherently opportunistic, i.e., SoSs are made "from what's available" [10]. Thus, before a concrete SoS architecture has been decided, the SoS engineer may want to choose, among several existing CSs, those ones that better contribute to achieve the planned mission with acceptable quality and costs.

Clearly, as the available CSs could provide many different functionalities that could be combined in various ways, the SoS engineer is facing here a space variability problem. To address this problem, instead of first designing the SoS architecture and then choosing among the available CSs the ones that best fit the design, as for example depicted by Cherfa et al. [6], we propose here an inverted approach, i.e., the SoS engineer opportunistically tailors the SoS design (of course relatively to non mandatory requirements) depending on the availability and cost of the functionalities offered by the candidate CSs. To the best of our knowledge, no existing methodology for SoS design works bottom-up considering the varying functionalities offered by the candidate CSs as an input for refining SoS architectural decisions.

Handling high-variability has been for years the subject of research in the software product line (SPL) domain [4]. Hence, rather than re-inventing from scratch new methods, we propose to look at existing knowledge and tools produced by SPL research, as we already speculated in a recent work [11] with reference to the challenges of SoS testing. In this paper we further develop this idea, and propose to leverage the variability and asset reuse concepts defined in the SPL domain to provide a design and testing approach called VANTESS (VAriability aware-desigN and TEsting of SoS) for the opportunistic engineering of directed or acknowledged SoSs. To this purpose we adapt the work by Nebut et al.'s [12,13,14] concerning variability modeling and test derivation: using this approach we can identify all the variation points relative to the available CS functionalities that lead to alternative architectural decisions, or SoS variants. Then, to support the SoS engineer in the selection of the most desirable SoS variant, we leverage a simplified version of the CBAM (Cost Benefit Analysis Method) method [15], which is a well-known practical strategy to support architectural decisions.

Summarizing we introduce the VANTESS approach that supports an SoS engineer in:

a. modelling the variation points induced by differing candidates CSs;

b. assessing the "desirability" of the resulting SoS variants;

c. deriving a simulation model of the selected solution;

d. generating a set of functional and robustness test cases.

In particular, for both a. and b. this paper provides own solutions, inspired by Nebut et al. [13] and Kazman et al. [15], respectively, but revised for the context of SoS design. Instead, for c. and d. we could apply the approach proposed by Nebut et al. [12,14] almost as is. Overall, this is the first work explicitly introducing and elaborating the concept of an opportunistic approach to early SoS design.

We demonstrate the approach by walking-through each of the steps over an educational SoS that has been developed and can be found at the VANTESS GitHub repository, available from https://github.com/edufysos/vantess, as a further contribution of this work. We have also conducted a focus group evaluation in which benefits and issues of the approach have been discussed within a group of six academic experts.

The paper is structured as follows: in the next section we briefly introduce SoS basic concepts; then in Section 3 we present some background material for the techniques we adopt as well as the application example used throughout the work. In Section 4 we present an overview of VANTESS describing its main phases, while in Section 5 we demonstrate its application. We report about the methodology and the results of the focus group in Section 6. Finally, we overview more closely related work in Section 7, and draw conclusions, also discussing limitations and future research directions, in Section 8.

2 | BASIC CONCEPTS

In this section we introduce some basic concepts related to SoS architectures and SoS variability management.

2.1 | SoS architectures

Even though a common definition for SoS does not exist, all authors agree that an SoS can be considered as a collection of pre-existing and/or independently owned and managed systems that cooperate to offer a service. The categorization of SoSs from the US Department of Defense[1] distinguishes four different SoS architectures according to how the CSs are organized to accomplish the SoS mission. These architectures are:

- Directed SoSs, which are assembled and centrally managed solutions conceived in order to satisfy a specified purpose. In this SoS architecture, the CSs have the ability to operate independently, but their normal operational mode is subordinated to the centrally managed purpose.
- Acknowledged SoSs, in which a central system exists that monitors the mission; the CSs retain self-control, but abide by the provided interaction guidelines to guarantee mission achievement.
- Collaborative SoSs, which are similar to acknowledged ones, with the difference that no central entity exists. The set of CSs working collaboratively needs well-defined interaction guidelines and responsibilities.
- Virtual SoSs, which lack a central management authority and an explicit shared purpose among the CSs; the CSs working under this architecture are offering their services without awareness of the mission to which they are participating.

These architectures reflect the behavior of CSs while accomplishing the joint mission. The same CSs may participate simultaneously to different SoSs, each one with a different architecture.

According to the guidelines of the U.S. Department of Defense[16], the core challenging activities of SoS Systems Engineering include: 1) translating SoS capabilities and objectives into high level requirements, and ii) understanding the individual systems and their technical context for identifying viable options to address SoS needs, also considering the impact of these options at system level.

The approach we propose in this paper can help address these challenges in the engineering of directed or acknowledged SoSs. As we describe in Section 4, VANTESS provides SoS engineers with a requirements model taking into account SoS space variability according to the functionalities offered by the available CSs. Moreover, VANTESS also allows for considering the impact of each solution in terms of its benefit and cost.

2.2 | Managing variability in SoS

Variability management includes a set of activities for representing variability in software artefacts, and leveraging it for building and evolving a family of software systems[17].As said, SoS variability can occur in terms of both time and space[4]. Variability in time occurs with the evolution of SoS. This evolution can take place through the entry, exit or evolution of a CS, and can lead the SoS to expose different behaviors over time. The variability in time is crucial in SoS management, but it is beyond the scope of this work.

Concerning space variability, the combination of several available CSs and their offered functionality can lead to different SoS configurations that fulfill the overall SoS mission in different ways. In other words, more candidate CSs can fulfill a certain individual mission, and, in turn, each CSs can accomplish this in different ways through its various features provided. Since an SoS can be composed of several individual missions that contribute to accomplishing the overall mission, the number of possible configurations can grow exponentially. Therefore, selecting a configuration that best fits the pre-established requirements is a challenge in SoS early design.

Handling variability is a key activity in SPL: it refers to the ability of an artifact to be configured, customized, or extended in order to derive different member products. The SPL research community has largely investigated the challenges related to variability management during the last two decades[4,17]. We see quite interesting convergences between SoS and SPL domains: both share some common principles, such as the large-scale reuse of existing artifacts and the abilities of (re)configuration and dynamic and fast customization. However, there are also divergences: in SPL the components are customized and integrated into a specific product containing the desired features, while in SoS, the CSs cooperate to accomplish a global mission. Also, an SPL component is used considering different configurations and features in each new product; in SoS, a CS is reused as is, usually without customization, but picking some of its offered features, according to the mission to be accomplished. Another difference concerns autonomy. While the fundamentals SoS characteristics are the operational and managerial independence of the CSs, in SPL, the components are generally provided by a central owner.

As we discussed in our previous workshop paper[11], a challenge of SoSs is that of having a system model expressing all variable functionalities of CSs, ensuring that only the required functionalities are included in the resulting SoS. For addressing this challenge, different approaches able to express the commonalities and variability of the products in SPL could be adopted. Several different variability models have been proposed, including feature models, decision models, orthogonal models, use cases models or domain specific models[18,19].

In this paper, for expressing SoS requirements variability, we propose a UML use cases-based variability model, which is obtained by leveraging and adapting an existing approach from the SPL domain[13], as we further explain in the next section.

3 | BACKGROUND

In this section we provide some background knowledge about: i) the approach we leverage and adapt from the SPL domain, ii) the CBAM heuristic supporting architectural decisions, iii) the mKAOS language used for the mission description, and finally iv) the SoS example we developed in a previous work and use here to show the applicability of our approach.

3.1 | Leveraging SPL variability models

In VANTESS we define a UML use cases-based SoS variability model. Specifically, we adopt and adapt Nebut et al.'s approach[13] for expressing SoS requirements variability. The main idea of Nebut et al.'s approach[13] is to define variability of SPL functional requirements by using UML use cases enhanced with parameters and contracts. The use cases parameters represent the inputs of the use cases, they can be actors or main concepts of the application. The contracts are first-order logical expressions on predicates. The predicates describe facts on the system and contain a name, and a (potentially empty) set of typed formal parameters that are a subset of the use cases parameters. They are either true or false. The operators are the classical boolean logic operators such as conjunction, disjunction, negation, implication (which is used to specify conditional contracts), and quantifiers (forall and exists). These contracts are expressed as pre and post conditions of the use case: the former represent the guards of the use case execution, the latter express the new values of the predicates after the execution of the use case.

Moreover, Nebut et al.'s approach defines UML tagged values for contracts, parameters, and use cases model elements. These UML tagged values are expressed as: VP{variant_list}, where VP is a variation point name and variant_list is a list of instantiations of the variation point. Those tags are a way to specify which parts of the requirements depend on a particular variant. If a tag is attached to a given model element, then this model element is taken into account only for the product selected by this tag, i.e., the product owning one of the variants specified in the tag, whereas a model element with no tag is taken into account for all the products. We refer to the original proposal by Nebut et al.[13] for a more detailed description and examples of this variability model.

Starting from the functional variation points defined at the SPL requirement level, Nebut et al.'s approach[13] allows for automatically deriving the requirements for a specific product, expressed as enhanced UML use cases. These product-specific enhanced use cases are then simulated by a use cases transition systems (UCTS) in which each transition is labeled with an instantiated use case. This simulation model is then used for deriving a set of test objectives, whereby a test objective is a sequence of instantiated use cases, namely a path in the UCTS. In order to derive an effective set of test objectives the following coverage criteria of the UCTS have been defined[20,12]:

- All Edges criterion (AE). Given a use case transition system ucts, this criterion is satisfied by a set of test objectives TOs iff each edge in the ucts is exercised by at least one test objective in TOs.
- All Vertices criterion (AV). Given a use case transition system ucts, this criterion is satisfied by a set of test objectives TOs iff each vertex v in the ucts is exercised by at least one test objective in TOs.
- All Instantiated Use Cases criterion (AIUC). Given a use case transition system ucts, this criterion is satisfied by a set of test objectives TOs iff each instantiated use case of the system is exercised by at least one test objective in TOs.
- All Vertices and All Instantiated Use Cases criterion (AV-AIUC). Given a use case transition system ucts, this criterion is satisfied by a set of test objectives TOs iff each instantiated use case of the system and each vertex in the ucts are exercised by at least one test objective in TOs.
- All Precondition Terms criterion (APT). Given a use case transition system ucts, this criterion is satisfied by a set of test objectives TOs iff each use case is exercised in as many different ways as there are predicates combinations to make its precondition true.

For each criterion an algorithm has been defined and implemented into a prototype tool that derives a set of test objectives satisfying it. The first two criteria AE and AV are traditional structural coverage criteria, the AIUC criterion aims at covering all the labels of the labeled transition system, AV-AIUC is a combination of AV and AIUC criteria, whereas APT represents a semantic criterion in which all the valuations making the precondition true are computed.

Nebut et al.'s approach[12] can also derive robustness test objectives aiming to detect robustness weaknesses of the system. For doing this, the adopted criterion is similar to the APT one, namely for each use case, all the shortest paths leading to each of the possible valuations that violate its precondition are considered[12]. Finally, from test objectives, test scenarios in the form of sequence diagrams can be derived starting from use case scenarios[14]. These test scenarios represent the messages sent to the system under test, but they are yet incomplete tests that cannot be executed with a test driver. The last step of Nebut et al.'s approach[12] is to derive from these test scenarios a set of test cases in the form of Java classes.

In VANTESS we leverage and adapt Nebut et al.'s approach for the modeling of SoS space variability through enhanced use cases. As we will show in Section 4.3, we define a variability notion tailored for specifying nice-to-have functional requirements in SoSs and derive from this variability model the requirements of each SoS variant. Starting from the so-specified requirements, a set of test cases for the selected SoS variant is finally derived as we will show in Section 4.6.

3.2 | Leveraging CBAM for SoS variant selection

When designing an SoS, the SoS engineer has to perform an analysis of costs and benefits coming with each considered configuration, in order to choose the better SoS variant to implement. This problem is related to software investments decision problem at system design level, for which many solutions have been proposed in the literature. Among them, we leverage and modify in VANTESS, the CBAM (Cost Benefit Analysis Method) method[15], which aims at improving the size of the design space supporting the stakeholders in making a good (not optimal) decision.

The main idea of CBAM is to identify a set of architectural strategies and evaluate their benefit and cost. The benefit is represented by the degree to which the architectural strategies support the different quality attributes of the system (performance, security, availability, usability, etc.). For instance, adopting a secure communication protocol in the system architecture could bring a relevant contribution to the overall system security. In CBAM, different stakeholders assign a quality attribute score to each quality attribute, then the benefit of each architectural strategy is computed in terms of its contribution to each quality attribute. A desirability metrics is computed as the ratio between the benefit and the estimated implementation cost of each architectural strategy, and is then used for ranking the architectural strategies. Kazman et al.[15] assume general cost estimation values (Low, Medium, High) assigned to each architectural scenario according to general cost factors such as elapsed time, shared use of critical resources, dependencies among implementation efforts.

The CBAM method has been applied to a large real-world project over several years[21] and lesson learned have been used to improve the method and solve issues mainly related to interpretation and differing understanding of the quality attributes among the stakeholders.

Falessi et al.[22] characterize existing decision making techniques for software architectural design by highlighting variability and commonalities among them; the considered techniques, among which CBAM, have been ranked according to their "level of susceptibility" to a specific set of difficulties that the software architect wants to avoid in the design of the system. The results of this study show that no decision-making technique is more (or less) susceptible than any other technique to the entire set of considered difficulties, then there is not a decision-making technique that is always better than any other one. However, the CBAM method that we leverage in our work showed a low susceptibility level in most of the difficulties that have been considered.

In VANTESS we leverage and adapt the main idea of CBAM for quantifying the costs and benefits of the different SoS variants. As we will discuss in more detail in Section 4.4, one main difference is that in VANTESS we do not consider two main sources of uncertainty present in the original CBAM application that are: i) variation in stakeholders' judgments (in VANTESS only one stakeholder is considered); and ii) the stakeholder's contribution score: thus, the contribution score of each SoS variant can be automatically computed as it is not dependent on the stakeholder judgment.

Another important difference of VANTESS with respect to CBAM consists in the cost estimation. Whereas in CBAM a general cost estimation value (Low, Medium, High) is assigned to each architectural scenario according to general cost factors such as elapsed time, shared use of critical resources, dependencies among implementation efforts, in VANTESS the cost of an SoS variant is computed in terms of the cost of the selected concrete CSs and is automatically computed by applying a simple greedy heuristics as we will detail in Section 4.4.

3.3 | mKAOS

mKAOS is a specialization of KAOS (A goal-oriented requirements engineering method) that aims to support SoS mission modelling[23]. It considers that a global mission can be successively refined into smaller missions until reaching the level of granularity of an individual mission, which matches the capabilities offered by the CSs. Thus, it is possible to assign individual missions to CSs and correlate global missions to emergent behaviors arising from the interactions between such CSs within the SoS.

The mKAOS language has six different models, the main one of which is the mission model. In this model, missions are structured as trees, in which leaf nodes represent individual missions, and non-leaf nodes represent global missions. Refinement links establish a refinement relationship between missions to refine a particular mission into other sub-missions. Figure 3 shows an example of an mKAOS mission model, where the blue rectangles represent the missions, the yellow circles represent the refinements of those missions, and the orange diamonds represent the abstract constituent systems. For the sake of space, we refer to Silva et al.[23] for further details.

3.4 | "Educational" SoS

In a previous work[24], we introduced EDUFYSoS, a "factory" of SoSs in the distance learning and educational domain that allows researchers in SoSs to instantiate different SoSs examples for experimenting their approaches. Accordingly, in this paper we use as an example an SoS instance belonging to that factory, which we refer to as the "Educational" SoS. Precisely, we develop here an enhanced version in order to show the applicability and usefulness of VANTESS.

In this section, we briefly describe the basic concepts of "Educational" SoS, which is used in some examples in Section 4 for facilitating the approach description and is then walked-through in Section 5.

The goal of "Educational" SoS is to allow university students to attend online courses and manage the classes and their assignments in an integrated way. Specifically, the actors involved in the "Educational" SoS are: i) students who should be able to learn contents related to a course appropriately; ii) employees belonging to the administrative staff who are in charge of providing the students with a list of courses and of facilities that allow them to register to the selected courses; iii) teachers who are responsible for managing their courses, scheduling the assignments and the online classes.

As depicted in Figure 1, the abstract CSs that have been identified for the "Educational" SoS are:

- Administrative Office System (AOS): this system should have the ability to manage information about courses, classes, students, and teachers at the university. An example of a system having this capability is the secretariat of the university.
- Learning Management System (LMS): this system should deliver online educational courses, handle the management of students and teachers in an online course, and finally enable teachers to create homework/assignments, assign them to students, trace students' activities, and report on their results.
- Calendar System (CS): this represents a time-management system providing the user with the ability to manage appointments, events, and deadlines.

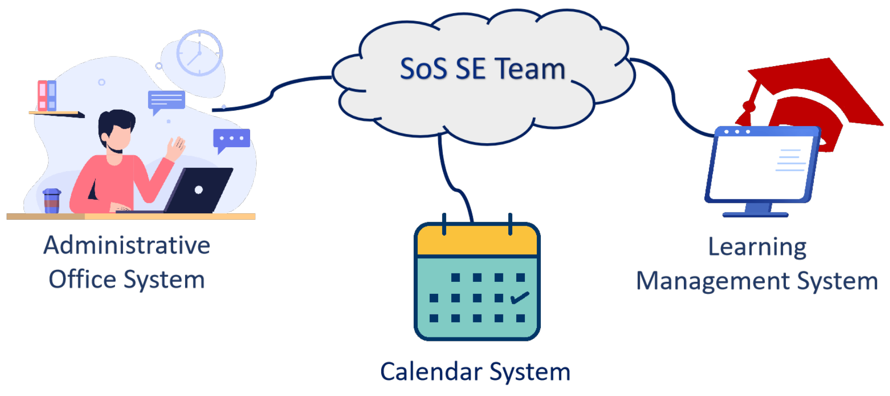

FIGURE 1 Overview of the "Educational" SoS.

These abstract constituent systems will be instantiated with concrete systems in application example presented in Section 5.

4 | OVERVIEW OF THE APPROACH

SoSs can have a very complex architecture since they may be constituted by several CSs that interact with each other. Many CSs may exist that expose similar functionalities, thus by variously composing such CSs, different SoS configurations, or SoS variants, could be derived. These SoS variants may present different functionalities, but at an abstract level they expose similar behaviors, namely they are able to accomplish a same main mission.

To master such potential space variability, an adequate design methodology is required that supports the SoS engineer in the choice of the CSs and their functionalities. Following such methodology, an approach should be used to assure that the SoS requirements are fulfilled and the SoS architecture can be easily maintained. The approach we propose, called VANTESS, goes in this direction by providing a variability-aware-design and testing approach of SoSs: the main novelty of the approach resides in its opportunistic management of space variability in early SoS design stage, by which the SoS architecture is tailored based on an appraisal of benefits and costs of the features offered by the available CSs.

In this section, we give an overview of VANTESS, and describe its main phases that are visualized in Figure 2. VANTESS supports the SoS engineer all along the SoS design and testing processes. Briefly, starting from the abstract definition of the SoS mission, a list of high-level functional requirements is derived, considering the functionalities offered by the available concrete CSs. These requirements are marked as must-have requirements, (i.e. necessary for the development of the desired SoS) and nice-to-have requirements (namely those that are desirable but not necessarily required for accomplishing the main goal of the SoS). From these high-level functional requirements a more formal variability model, specified through enhanced use cases, is derived. By means of specific tagged values this variability model expresses which are the must-have requirements and the nice-to-have requirements. Leveraging this variability model, different SoS variants are derived, for instance by means of combinatorial approaches over the different variability attributes of the variability model. Must-have requirements must be fulfilled by all SoS variants, while the nice-to-have requirements will belong to specific variants. Then, VANTESS allows the SoS engineer to assess the desirability of different SoS variants (trading off among benefits and costs). Finally, for the selected SoS variant, VANTESS allows to derive a set of executable test cases through SoS model simulation and test objective instantiation.

In the following sections, we describe in detail the phases of VANTESS that are: i) SoS mission definition (phase 1 in Figure 2); ii) CSs recognition (phase 2 in Figure 2); iii) modeling variability of SoS requirements (phase 3 in Figure 2); iv) assessing desirability of SoS variants (phase 4 in Figure 2); v) simulation of the selected SoS variant (phase 5 in Figure 2); and finally vi) generation of test cases for the selected SoS variant (phases 6, 7, and 8 in Figure 2). Moreover, Figure 2 also shows which phases are automatically, semi-automatically or manually performed.

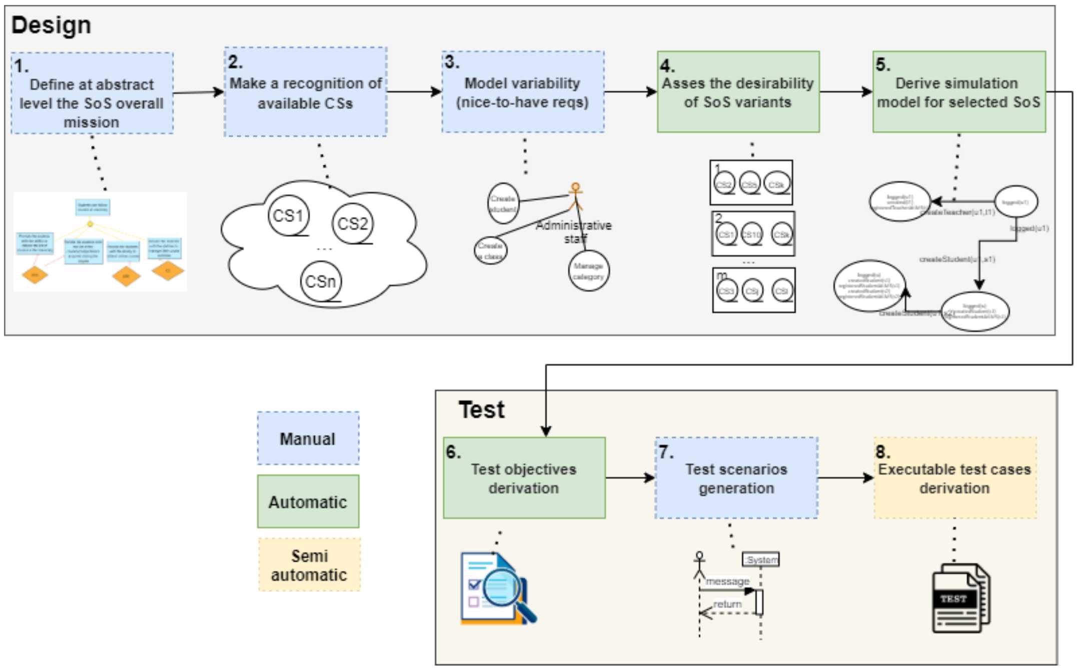
Design
1. Define at abstract level the SoS overall mission
2. Make a recognition of available CSs
3. Model variability (nice-to-have reqs)
4. Asses the desirability of SoS variants
5. Derive simulation model for selected SoS
CS1
CS2
...
CSn
Create student
Create a class
Administrative staff
Manage category
createTeacher(u1,t1)
logged(u1)
createStudent(u1,s1)
Test
6. Test objectives derivation
7. Test scenarios generation
8. Executable test cases derivation
System
message
return
TEST
Manual
Automatic
Semi automatic

FIGURE 2 Overview of the VANTESS approach.

4.1 | Overall SoS mission definition

In the first phase, the overall SoS mission is defined. Although at this stage the mission is still abstract, we can reasonably assume that the SoS engineer (together with the application domain expert) can determine the high-level requirements for the CSs to be involved and the abstract functions that these systems must provide. With this information, an abstract mission model can be defined. This definition can be done using differing notations, from informal ones, such as geometric figures, up to even formal languages. In this work, we chose to use the mKAOS [23] language, described in Subsection 2.2, as it is one of the pioneering languages to specify SoS missions, and supports the description of missions and their refinements until reaching the CSs level.

The model we define in this phase is not constituted by the actual CSs that will cooperate in the SoS but rather by the "types" of CSs that the SoS needs. Hence, at this stage we refer to abstract CSs rather than to concrete systems. These abstract CSs will be then instantiated with concrete CSs in the next steps of VANTESS. In the original conception of mKAOS, a high level mission is then refined in sub-missions that are associated with the capabilities that the CSs must have in order to achieve the referred SoS mission. However, using mKAOS it is not possible to define SoS variability nor to associate more than one CS to a sub-mission, as we aim to do in VANTESS.

4.2 | Constituent systems recognition and requirements definition

The aim of the CSs recognition phase is that of associating to the abstract CSs defined in the previous step a set of concrete existing CSs that could be available when the SoS is designed. In other words, to enhance re-use we prospect somehow that the design of an SoS accomplishing a given high-level mission proceeds bottom-up depending on the available CSs. Among the abstract CSs, some could be fixed due to practical constraints, other could be instantiated by using different concrete systems that provide the searched functionalities. For instance, referring to the "Educational" SoS, introduced in Section 3.4, we imagine that the AOS is fixed because the university will adopt the administrative system that is in place; concerning the LMS different concrete instances could be considered, namely Moodle, FullTeaching, or Google Classroom; finally examples of two available concrete systems having the capability of Calendar System could be Google Calendar or Yahoo Calendar.

Moreover, in this phase, considering the functionalities offered by the available concrete CSs, a list of high-level functional requirements for the desired SoS is defined. In VANTESS we require that the SoS engineer distinguishes these requirements are into two main categories: those that are considered necessary for the development of the desired SoS, or must-have requirements, and those that are desirable but not necessarily required for accomplishing the main goal of the SoS, or nice-to-have requirements.

4.3 | SoS requirements modeling based on enhanced use cases

The goal of this phase of VANTESS is to allow the SoS engineer to define a more formal requirements model, starting from the list of high-level functional requirements defined in the previous phase. This model must be able to capture those variability aspects that exist beyond the SoS conception, so to specify which are the must-have requirements that must be common to all SoS variants and which are the nice-to-have requirements that belong to particular variants. The task of modelling the possible set of variants appears similar to the specification of variability in software product lines. Accordingly, to manage the variability aspects in the SoS requirements modeling, we borrow and adapt Nebut et al.'s approach [12,13], based on enhanced use cases, that was originally proposed to define and manage the requirements for a software product line. Use cases are a well-known and widely used means to express system functional requirements. Following the proposal of Nebut et al. [12,13], in VANTESS we adopt UML [1] use cases enhanced with parameters and contracts on these parameters. As an example, the following enhanced use case (abbreviated as UC):

```
    UC enroll student into a class (s: student; cl: class)
pre created(s) and created(cl)
post enrolledStudent(s, cl)
```

represents a functional requirement for the "Educational" SoS, expressing that a student should be able to enroll in a class. This use case is parameterized by the student who wants to enroll in a class and the class itself. Use case contracts, belonging to pre and post conditions, are expressed in UC as predicates specifying a name and a set of parameters. For instance, the predicate created(s) in the precondition is true when the student has been created, false otherwise.

[1] https://www.omg.org/spec/UML/About-UML/

By using enhanced use cases modeling, VANTESS allows the SoS engineer to specify commonalities and variability of SoSs, namely to specify which parts of the requirements are common to different SoS variants and which ones depend on a particular SoS variant. For defining variability, as in Nebut et al.[13], we use tags (UML tagged values) expressed in the form: VPName{variant_list}, where VPName is a variation point name and variant_list is a list of instantiations of the variation point, that we call variation point attributes. A variation point can be associated with one or more UCs. For the purposes of SoS modeling, we simplify the application of Nebut et al.'s approach considering for each variation point only two possible variation point attributes. Precisely, given a variation point VP_i, expressed in the enhanced use case model, we define the set of the associated variation point attributes to VP_i, called $VPAttr_i$, with i=1,..n, as follows:

$$VPAttr_i = \{VP_i\_True, VP_i\_False\}$$

This simplification of the Nebut et al.'s approach does not introduce a limitation in the presented approach since this is enough to express as variability points the nice-to-have functional requirements of the SoS, namely the functional requirements that could be present or not in a particular SoS variant. For instance, referring to the "Educational" SoS, in the use case

UC set assignments (t:teacher; cl: class; a:assignment; l:lesson) VPSendNotification(SendNotification_True, SendNotification_ False)

the tag VPSendNotification (SendNotification_True, SendNotification_False) means that a notification could be sent or not to the students when the teacher sets an assignment related to a lesson. The set of associated variation point attributes to the variation point VPSendNotification is
$VPSendNotificationAttr = \{\text{SendNotification\_True, SendNotification\_False}\}$
Given n variation points in the enhanced use case model, and VPA = $\cup VPAttr_i$ with i= i..n, we define an SoS variant SoS_k in terms of the set of the true variation point attributes it contains [2], as follows:

$$SoS_k = \{\text{vpa}_i \in \text{VPA} | \text{ vpa}_i = VP_i\_True \wedge 1 \leq i \leq n \}$$

For instance, from the enhanced use case "UC set assignments (t:teacher; cl: class; a:assignment; l:lesson)" described above, we can specify, according to the variation point VPSendNotification, an SoS variant containing the variation point attribute SendNotification_True that represents the nice-to-have functional requirement that allows to send a notification when an assignment is set.

For brevity, we leave out of the scope of this paper to investigate a procedure to parse all the variation points of the enhanced use case model for deriving all the possible SoS variants in terms of their variation point attributes. Rather we are interested into a method to select among them one variant that is the most "desirable", as we describe in the next phase.

4.4 | Assessing the desirability of SoS variants

The goal of this phase is to assess the different SoS variants in terms of their desirability metrics, computed as the ratio between the benefit and the cost of each SoS variant. Since each SoS variant is expressed in terms of its variation point attributes, this desirability metric will support the SoS engineers in the choice of the right set of variation point attributes, corresponding to the set of nice-to-have functional requirements of the SoS to be implemented. We presented in Section 3 the main steps of CBAM. Here, we want to show how VANTESS adapts the main idea of CBAM for quantifying the costs and benefits of the different SoS variants, and then assign a desirability value to each SoS variant.

The SoS variants derived in the previous phase of VANTESS implement different variation point attributes. Each variation point attribute may bring some benefits to the desired SoS, but on the other hand its implementation could also undergo some risks and costs. If the number of variation point attributes expressed in the use cases model grows, the SoS engineer might have to analyze an overwhelming number of SoS variants. By applying the CBAM method, we can automatically calculate the desirability metric for each SoS variant. Specifically, this is done in two steps:

[2]We are interested in defining an SoS variant in terms of the true variation point attributes it contains, corresponding to the nice-to-have functional requirements it implements assuming that by construction each SoS variant will also implement the must-have functional requirements defined in the enhanced use case model.

i) compute the benefit of each SoS variant, representing the SoS variant capacity to offer some nice-to-have functional requirements rather than others. The benefit of an SoS variant SoS_i is defined as:

$$Benefit(SoS_i) = \sum_j (Cont_{ij} * VPAScore(j)), j : 1...\#VPA \tag{1}$$

where:

$VPAScore(j)$ is a score assigned by the SoS engineer[3] to the variation point attribute $VPA_j \in$ VPA, representing the relative value of importance of having this variation point attribute in the desired SoS. The SoS engineer assigns the scores to all the variation point attributes of an SoS variant so that they sum up to 100.

$Cont_{ij}$ is the contribution of SoS variant i to the variation point attribute VPA_j defined as:

$$Cont_{ij} = \begin{cases} 1, \text{ if } SoS_i \text{ contains } VPA_j \\ 0, \text{ otherwise} \end{cases} \tag{2}$$

We use the benefit function as expressed in Equation 1 since we simply want to model the benefit of an SoS variant in terms of nice-to-have functional requirements it offers and then in terms of the variation point attributes it contains. Each variation point attribute counts 1 (if the variation point attribute belongs to the SoS variant) or 0 (if the variation point attribute does not belong to the SoS variant) and has a score representing its relative importance with respect to the other variation point attributes defined in the SoS model.

Note that, contrariwise to CBAM[15] where the stakeholders have to manually assign a contribution score to each architectural decision, in VANTESS the definition of the contribution score of each SoS variant is very simple (1 or 0) and less refined than in CBAM but it can be automatically derived applying the above formula (2) avoiding the manual assignment from the stakeholders.

The main limitation of VANTESS with respect to the original CBAM decision-making technique is the number of the involved stakeholders. In VANTESS, for aim of simplicity we consider only one stakeholder, but the approach could be extended to consider more than one stakeholder. Moreover, another limitation of VANTESS with respect to CBAM method is that VANTESS does not consider dependencies among SoS variants, since we can confidently assume that each one is independent from the other ones.

Finally, a restriction of the benefit function adopted in VANTESS is that of considering, as in CBAM, the variation point attribute scores as single values into a defined range (for instance a score such that the sum of the scores is 100). More complex measures for the variation point attributes could be defined depending for instance on the application scenarios, or using utility level associated to the variation point attributes as in Moore et al.[21].

To illustrate the application of VANTESS to compute the benefit of an SoS variant, let us consider the following dummy example: i) #VPA is 10; ii) the VPAScore(s) associated to VPA_1, VPA_2, VPA_3, VPA_4, VPA_5, VPA_6, VPA_7, VPA_8, VPA_9, VPA_{10} are 10, 5, 10, 5, 5, 10, 20, 10, 10, 15, respectively; this means VPA_7 corresponds to the most important nice-to-have functional requirement to consider in the SoS. Let us consider the following SoS variant: $SoS_{ex} = \{VPA_1, VPA_3, VPA_5, VPA_7, VPA_9\}$, containing the variation point attributes: VPA_1, VPA_3, VPA_5, VPA_7, VPA_9. According to Equation 1, then we have that $Benefit(SoS_{ex}) = 55$.

ii) compute the expected cost associated to the implementation of each SoS variant. In VANTESS the cost of an SoS variant is computed in terms of the cost of the selected concrete CSs that could implement all variation point attributes contained in an SoS variant. Given an SoS variant SoS_i, this cost is defined as:

$$Cost(SoS_i) = \sum_k Cost(CS_k), k = 1...m, \tag{3}$$

where $\{CS_1, ...CS_m\}$ is a set of concrete selected CS instances implementing the variation point attributes of SoS_i.

Contrariwise to CBAM[15] where the stakeholders have to manually assign a cost to each architectural decision, in VANTESS the assignment of the cost to each SoS variant is derived automatically by applying a simple heuristics. Precisely, we adopt an additional

[3]Note that in CBAM, different stakeholders assign a quality attribute score to each quality attribute, the average of these scores representing the relative importance of each quality attribute. In VANTESS, for aim of simplicity we consider only one stakeholder, but the approach could be extended to consider more than one stakeholder.

greedy ordering heuristics[25] [4] showed in Algorithm 1 to automatically select the set of CS instances implementing all the variation points attributes of a given SoS variant.

Algorithm 1 CSs Selection (greedy additional heuristic)

```
1: input: $S$                                   ▷ The set of CSs along with their cost
2: input: $VPA$                                 ▷ list of variation points attributes to be covered
3: input: $coverageinfo$                        ▷ list of variation points attributes covered by each CS from S
4: input: $costinfo$                            ▷ cost of each CS
5: output: $S'$                                 ▷ a subset of CSs, with which to implement the target SoS variant
6: $S' \leftarrow []$                           ▷ S' is initialized as an empty list
7: while thereAreUncoveredVPA(S, VPA, coverageInfo, costInfo) do
8:     $selectedCS \longleftarrow getNextCS(S, VPA, coverageInfo, costInfo)$   ▷ selects the CS with the minimum cost that covers the highest number of uncovered VPA
9:     $add(selectedCS, S')$
10:    $updateUncoveredVPA(VPA, selectedCS)$    ▷ removes the VPA covered by the selected CS from the list of uncovered VPA
11: end while
```

The main idea is to repeatedly select the CS instances that cover the maximum number of uncovered variation points attributes until all variation points attributes of an SoS variant are covered. If more than one CS instance cover the same number of uncovered variation points attributes, the CS instance with the minimum cost is selected.

As an example, let us consider the set of variation point attributes VPA_1, VPA_2, ...VPA_{10} (first column of Table 1) and the same SoS_{ex} variant containing a subset of these variation point attributes, marked as yes in the second column of Table 1. The table also shows a set of concrete CS instances (CS_1, CS_2, CS_3, CS_4, CS_5), each one displayed along with its cost in a range [1..10], and the variation point attributes it covers. The first CS that the Algorithm 1 selects is the one that achieves the highest coverage of variation point attributes. When applied to the example in Table 1, CS_1 is selected that covers 3 variation point attributes (VPA_1, VPA_5, VPA_7) out of the 5 ones contained in SoS_{ex} variant. Then, it looks for the next CS that achieves the highest coverage with respect to the yet to be covered variation point attributes (VPA_3 and VPA_9), and CS_3 is selected that is the only one covering VPA_3. For the next choice - the CS that enables to cover VPA_9 - two CSs are tied (CS_4 and CS_5), CS_4 is selected since it is the one with lower cost. According to Equation 3, the cost of SoS_{ex} variant is 7 (i.e., the sum of the costs of CS_1, CS_3 and CS_4).

TABLE 1 Variation point attributes coverage achieved by concrete CS instances

	SoS_{ex}	CS_1	CS_2	CS_3	CS_4	CS_5
		C=2	C=1	C=4	C=1	C=3
VPA_1	yes	✓				
VPA_2						
VPA_3	yes			✓		
VPA_4						
VPA_5	yes	✓	✓			
VPA_6						
VPA_7	yes	✓				
VPA_8						
VPA_9	yes				✓	✓
VPA_{10}						

Finally, the Desirabiliy of an SoS variant is computed as:

$$Desirability(SoS_i) = Benefit(SoS_i)/Cost(SoS_i) \tag{4}$$

For instance, the desirability of SoS_{ex} is 55/7.

[4]This heuristic has been proven to have a high cost-effectiveness ratio for white-box coverage-based test selection in software testing.

After computing the desirability of all the SoS variants, obtained from the enhanced use case model, the SoS engineer can rank the SoS variants according to their desirability metric and only for the SoS variants with high desirability he/she will perform more accurate cost estimation before implementation adopting for instance more complex cost models[26].

For sake of clarity, for assessing the desirability of each SoS variant we assume that all variation point attributes that are present in an SoS variant (and then the corresponding nice-to-have functional requirements) are independent from each other. We plan in future work to extend VANTESS in order to address more realistic scenarios in which dependencies among the nice-to-have functional requirements can be considered.

4.5 | SoS variant simulation

Once the desired SoS variant is selected, the VANTESS approach moves to the testing stage. The approach in fact allows to specify a set of test cases for assessing the functional requirements.

The generation of test cases relies on the use cases simulation for expressing the states of the system. It is beyond the aim of this work to investigate the effectiveness of existing approaches to model the states of a system. We follow Nebut et al.'s approach[12,13] that proposes a step for use cases model simulation. Specifically, according to Nebut et al.'s approach, a simulation model can be built by considering an initial state and an instantiation of the functional requirements expressed in the enhanced use cases of an SoS variant. For each use case, a set of instantiated use cases (called IUC) is obtained by replacing the formal use case parameters with all the possible combinations of their values. Similarly, a set of instantiated predicates is obtained by replacing the formal parameters of a predicate by all the possible combinations of their values, where for testing purposes a bounded set of arbitrary elements is considered. As an example, referring to the "Educational" SoS, if we suppose to have two students (s1 and s2) and a class (cl1), the enhanced use case

```
    UC enroll student into a class (s: student; cl: class)
pre created(s) and created(cl)
post enrolledStudent(s, cl)
```

is instantiated into two use cases "enroll student into a class(s1, cl1)" and "enroll student into a class(s2, cl1)", whereas the predicate "created(s)" is instantiated as "created(s1)" and "created(s2)", respectively.

Given a current state of the simulation model, an instantiated use case can be executed if its precondition is implied by this current state; then the new current state after the execution of the instantiated use case is a modification of the current state so that the post condition of the instantiated use case becomes true. The exhaustive use cases simulation is represented by a Use Case labeled Transition System (UCTS) in which each state represents a state of the SoS variant and the transitions among states are represented by instantiated uses cases. For the sake of space, we do not discuss here about the complexity of UCTS, but refer to the work by Nebut et al.[12], where this modelling formalism and the algorithms used for deriving and processing the graph are discussed in detail, along with potential limitations.

By exploring the simulation model, in this phase the SoS engineer can analyze all the possible reachable states of an SoS variant and check which are the valid functionalities of this SoS variant, namely which are the use cases that can be executed in each state. Finally, the SoS engineer can verify that the overall SoS variant behavior corresponds to the expected one, or detect inconsistencies and mistakes in the requirements specification.

4.6 | Test cases derivation

In the last phases, for a selected SoS variant, VANTESS allows to derive a set of test cases. The test cases generation is articulated in three consecutive main steps: i) derivation of test objectives; ii) generation of test scenarios; and iii) derivation of executable test cases.

Test objectives derivation

In this first step, considering the UCTS obtained for each SoS variant as described in Section 4.5, VANTESS allows to derive a set of test objectives. A test objective is defined as a valid sequence of instantiated use cases in the UCTS, beginning from the initial state. The notion of a valid sequence of instantiated use cases corresponds to the classical notion of a path in a graph, in which the first vertex corresponds to the initial state. The generation of test objectives is performed according to a given UCTS coverage criterion.

Among the available UCTS coverage criteria presented by Nebut et al.[20,12], and briefly described in Section 3.1, as in Nebut et al.'s approach, also in VANTESS we adopt the All Precondition Terms (APT) criterion for deriving test objectives. APT guarantees that all the possible ways to apply a use case (then a requirement) are exercised, namely that each instantiated use case of the UTCS is exercised according to all the predicate combinations that make its precondition true. We apply the algorithm presented in[12] for deriving, from the UCTS of the desired variant, a set of functional test objectives, satisfying this criterion. However, other coverage criteria could be used bringing to a different set of test objectives. Future work could investigate the most effective coverage criterion in terms of testing effectiveness.

The derived functional test objectives are represented by valid sequences of instantiated use cases.

Moreover, VANTESS allows for also deriving robustness tests for the desired SoS variant. [5] The simple idea beyond robustness test cases derivation is that of correctly exercising the SoS variant up to a given point, and then applying a not foreseen action in order to obtain a robustness test. This idea leverages Nebut et al.'s proposal of using the generated UCTS as an oracle for robustness tests and then generate invalid paths, namely paths leading to an invalid application of a use case. An invalid path is a path derived by the valid application of the first n-1 IUCs and then by the application of an invalid instantiated use case (IUC_n) (that means that IUC_n is applied if its precondition is false after the application of IUC_{n-1}). Then, robustness test objectives are derived as invalid sequences of instantiated use cases, namely use cases sequences containing an invalid application of an instantiated use case. We will present in Section 5 an extract of the simulation model for the "Educational" SoS together with several test objectives derived from this simulation model.

Test scenarios generation

The test objectives represented by sequences of instantiated use cases are not executable test cases in the sense that they cannot be directly executed on the SoS code. Following Nebut et al.'s approach[12], VANTESS allows to derive application test cases from the test objectives. The main idea is to replace the instantiated use cases in the test objectives with instantiated use case scenarios, expressed by UML sequence diagrams. Specifically, these sequence diagrams illustrate how an actor stimulates the system, and how the system responds.

Each sequence diagram can represent a nominal or exceptional scenario, whereby the former corresponds to a functional test in which the SoS is exercised to test a specific functionality and the latter corresponds to a robustness test in which the SoS is exercised in order to raise an exception or error message. We refer to Nebut et al.[12] for the specific procedure used for translating test objectives into test scenarios.

Executable test cases derivation

From the test scenarios above described, a set of executable test cases must be derived. The sequence diagrams can contain parameters as well as pre and post conditions that could include the same parameters as in the corresponding use cases. A large number of scenarios per use case can imply a combinatorial explosion of test cases to be executed. Nebut et al.[12] present a prototype tool that generates test cases, leveraging the test scenarios, in the form of Java classes and supports the execution of these tests. In this paper, for sake of demonstration we just provide in Section 5, some examples of generated test scenarios manually derived from a set of test objectives and then we show the implementation of a test case for the real instantiation of the desired SoS.

5 | APPLICATION OF VANTESS TO "EDUCATIONAL" SOS

In this section, we show the usefulness of VANTESS in the design and testing of the "Educational" SoS introduced in Section 3.4. We describe the applicability of VANTESS to this SoS all along the main steps of the approach described in Section 4. As a result, a new enhanced and fully instantiated design for "Educational" SoS is also provided and made available at the VANTESS GitHub repository, available at https://github.com/edufysos/vantess [6].

5.1 | "Educational" SoS mission diagram

The main goal of "Educational" SoS is to allow students to attend online courses at a given university and to manage their online activities in an integrated manner. Thus, the most abstract mission defined by the SoS engineer is "Students can follow courses at university", which must be achieved by "Educational" SoS through the collaboration of its constituent systems. This more general mission was refined into four concrete sub-missions: "Provide the students with the ability to choose the list of courses in the University"; "Provide the students with the list of the courses/competences acquired during the degree"; "Provide the students with the ability to attend online courses";

[5]Robustness testing would include the testing of exceptional scenarios of the use cases, whereas by applying the APT criterion we test the nominal scenarios.

[6]This new example is provided inside the EDUFYSoS organization on GitHub (https://github.com/edufysos).

"Provide the students with the ability to manage their course activities". These submissions need to be provided as capabilities by different constituent systems. For each required capability, we assigned a type of abstract constituent system, namely AOS, LMS, and CS. These abstract systems will be further discussed in Section 5.2. Figure 3 shows the "Educational" SoS mission diagram using the mKaos notation.

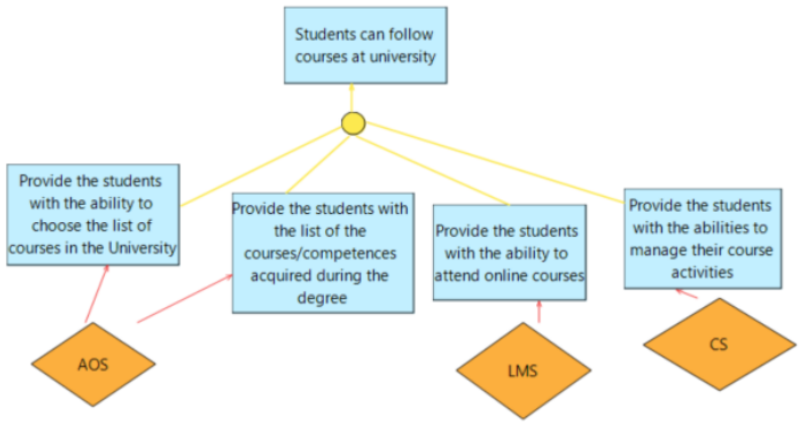
Students can follow courses at university
Provide the students with the ability to choose the list of courses in the University
Provide the students with the list of the courses/competences acquired during the degree
Provide the students with the ability to attend online courses
Provide the students with the abilities to manage their course activities
AOS
LMS
CS

FIGURE 3 "Educational" SoS mission diagram.

5.2 | Recognition of available CSs for "Educational" SoS

As stated in Section 3.4, for accomplishing the mission of the "Educational" SoS, the SoS engineer needs the following three different types of CSs: AOS, LMS, CS.

In this recognition phase, VANTESS allows the SoS engineer to associate to each CS type, a list of concrete available CSs implementing the functionalities of the abstract corresponding CSs. Moreover, the SoS engineer during this recognition of the available concrete CSs establishes which CSs are fixed and which can be instantiated with different concrete CSs. In the "Educational" SoS, AOS was fixed and implemented by means of RosarioSiS[7]. RosarioSiS is an open-source project promoting a management information system for educational institutions. Concrete instances of LMS can be offered by Moodle, FullTeaching, and Google Classroom whereas CS could be implemented using Google Calendar or Yahoo Calendar. We briefly describe below the main functionalities of these concrete systems referring to available documentation for more details:

- RosarioSiS is a free and open-source School Management System (SMS) whose main functionalities are managing teachers, students, classes, and grades. Also, it has a plugin that allows integration with Moodle. RosarioSoS is developed using the PHP language and PostgreSQL database. Additional information can be obtained at www.rosariosis.org;
- Moodle (Modular Object-Oriented Dynamic Learning Environment) is a free software for learning management. With Moodle, it is possible to create virtual learning environments, make material available offline, and assign tasks to students. It is developed in PHP and uses the MySql database. More information can be obtained at https://moodle.org/;
- FullTeaching is a web application that allows to make synchronous lessons. It leverages OpenVidu capabilities for real-time multimedia communications. It also allows the creation of courses and classes and the asynchronous communication, then providing content available offline. It is developed in Java and Angular and uses MySQL as a database. More information is available in the FullTeaching repository available at https://github.com/pabloFuente/full-teaching;
- Google Classroom is a free service from Google for managing teaching activities. It allows a teacher to create classes and enroll students. Other features are the management of activities, in which the teacher can make material available online and create assignments for his/her students. Also, it has integration with other Google services, such as Google Meet and Google Calendar. More information can be accessed at https://edu.google.com/products/classroom/;

[7]see at https://github.com/francoisjacquet/rosariosis

- Google Calendar is a free calendar service from Google. It is possible to create and schedule meetings and events and sharing them with other people. More information can be seen at https://developers.google.com/calendar;
- Yahoo Calendar is another free calendar service offered by Yahoo!. Like in Google Calendar, it is possible to create and schedule meetings and events, as well as sharing them with other people. More information can be seen at https://www.calendar.com/yahoo-calendar-guide/.

Leveraging the knowledge about the functionalities of these concrete CSs, in this step the SoS engineer defines also a list of requirements for "Educational" SoS, by specifying the desirability level of each requirement, namely which are the requirements that "Educational" SoS has to fulfill (they are tagged as must-have) and which ones the "Educational" SoS could fulfill (they are tagged as nice-to-have). Table 2 shows a subset of natural language requirements for "Educational" SoS, tagged with their desirability level. For instance, if creating a lesson is a must-have requirement (R5) for "Educational" SoS, creating a synchronous lesson (R7) is a nice-to-have one.

TABLE 2 A subset of requirements for "Educational" SoS

Requirement ID	Requirement Description	Desirability Level
R1	Create a student if it is not yet created	must-have
R2	Create a teacher if it is not yet created	must-have
R3	Create a class if it is not yet created	must-have
R4	Enroll a student in a class	must-have
R5	The teacher will be able to create a lesson	must-have
R6	Assign the course to a category if the category exists	nice-to-have
R7	The teacher will be able to create a synchronous lesson	nice-to-have
R8	The teacher will be able to open a chat	nice-to-have
R9	The teacher will send lesson material to students	must-have

5.3 | "Educational" SoS requirements model

Starting from the list of high level requirements as defined in Section 5.2, the SoS engineer provides a description of "Educational" SoS requirements in the form of UML use cases as in Figure 4. The "Educational" SoS allows the delivering and learning of contents as well as course administration in an integrated way. When students, teachers, courses and classes are created they are registered both at AOS and LMS. The students can request to enroll in a class and their enrollment is registered at both AOS and LMS, they can attend a lesson in the online learning system, enroll in a chat, and see all activities assigned to them in their calendar. The teacher is responsible for managing his/her classes, scheduling the assignments and the online lessons. People belonging to the administrative staff are in charge of creating students and teachers, providing the students with the ability to register to a course and to enroll in a class.

Table 3 shows a description of the enhanced use case model of "Educational" SoS in which, parameters, contracts, dependencies among use cases and variability points are expressed as described in Section 4.3. These use cases contain parameters: for instance the use case UC_1, is parameterized by the administrative staff in charge of creating the student and the created student, whereas in the use case UC_5 the parameters are the student who want to enroll in a class and the class itself. In these use cases the contracts, namely logical expressions on predicates involving these parameters, are defined as pre and post conditions. For instance the use case UC_4 "create class(u:administrativeStaff; c:course; t:teacher; cl:class)" requires that the class is created (Object Oriented Programming section A is a class example) if the administrative staff is logged in the system, the corresponding course (Object Oriented Programming course) and the teacher are created at AOS and LMS. After performing the class creation, the class is created, this means that it is registered at AOS and LMS and the teacher is enrolled into the class. By means of pre and post conditions, some dependencies arise between use cases. For instance, the student can enroll in a class (see use case UC_5) only if he/she has been created at AOS and LMS (use case UC_1).

The enhanced use case model of Table 3 expresses the must-have and nice-to-have functional requirements of "Educational" SoS defined during the recognition phase as in Section 5.2. The nice-to-have functional requirements are expressed in the enhanced use case model using variation points and variation point attributes, as described in Section 4.3. In particular, the enhanced use case model of Table 3 shows four nice-to-have functional requirements of "Educational" SoS that are:

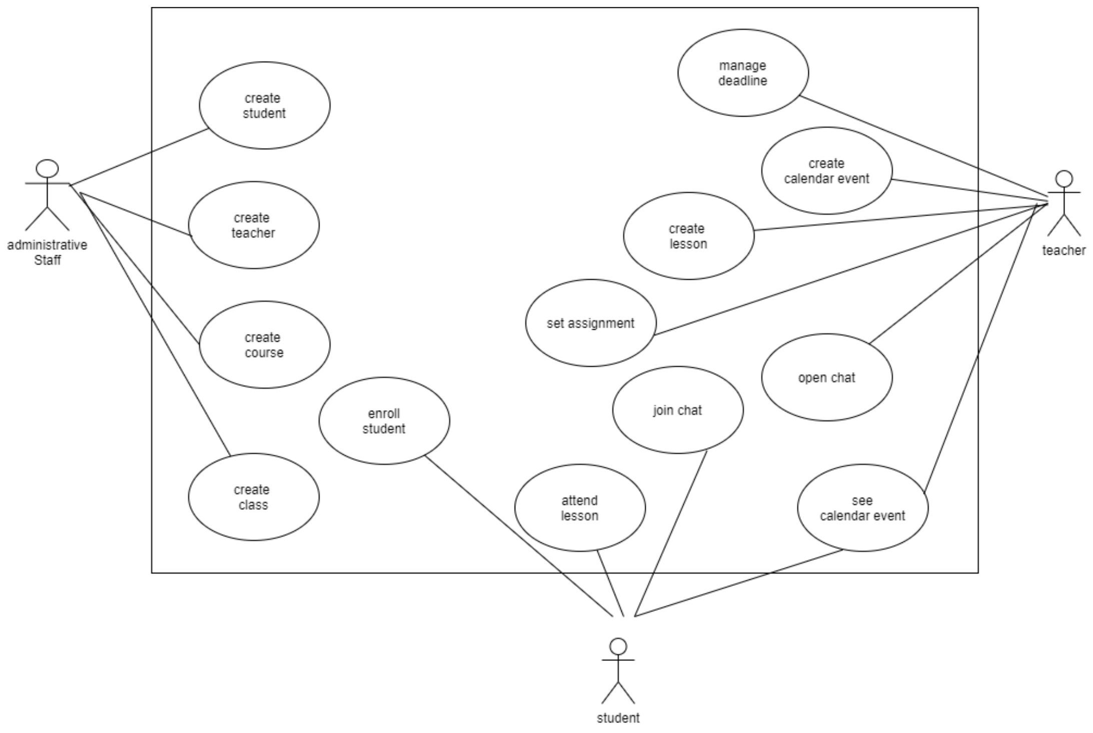

FIGURE 4 “Educational” SoS use case diagram.

- VPSettingCategory. The use case UC_3 of Table 3 requires that a course (for instance Object Oriented Programming course) is created if the administrative staff is logged in the system and the course has not been already created. This is a must-have functional requirement of the system. When creating a course the system could offer also the possibility to associate a category (for instance Computer Science category) for this course. This is a nice-to-have functional requirement of “Educational” SoS that is expressed by tagging the use case UC_3 with the variation point VPSettingCategory(SettingCategory_true, SettingCategory_false). The variation point attribute VPSettingCategory(true) allows to select the SoS variant owning this functionality. In this variant, when a course is created, a category is set for that course. Note that if a variation point attribute is associated with a pre or post condition, the fulfillment of the condition is required only when the variation point attribute holds. For example, again in UC_3 both post conditions $created(cat) if not created(cat)$ and $set(c, cat)$ are required only for those SoS variants owning the above described nice-to-have capability.

- VPSynchronousLesson. This nice-to-have functional requirement of “Educational” SoS deals with the possibility to manage synchronous and asynchronous lessons, expressed in UC_6 and UC_7 of Table 3 with the variability point VPSynchronousLesson(SynchronousLesson_ true, SynchronousLesson_ false). The variation point attribute VPSynchronousLesson(true) allows to select the SoS variant in which the teacher can schedule a synchronous lesson (namely a video conferencing), setting a date (UC_6 use case) and opening a chat (UC_8 use case) whereas the student can attend the lesson by joining the video conferencing (UC_7 use case) and the associated chat (UC_9 use case). Note that, for aim of simplicity, in this use cases model we assume that the chat management is a must-have functionality associated to the synchronous lesson.

- VPSendNotification. This functionality deals with the possibility of sending a notification when an assignment is set. Specifically, the UC_{10} use case expresses the must-have functionality of setting an assignment. The variation point VPSendNotification(SendNotification_true, SendNotification_false) expresses the nice-to-have functionality of sending a notification when the assignment is set. The variation point attribute VPSendNotification(true) allows to select the SoS variant in which the teacher sends a notification with the assignment (UC_{10} use case) and a notification of the deadline variation for that assignment (UC_{11} use case).

- VPImport-ExportCalendar. This functionality allows the user to export the course calendar and import it in his/her personal agenda. As specified in UC_{12} use case, the system must allow the teacher to set a calendar event. The variation point VPImport-ExportCalendar(Import-ExportCalendar_true, Import-ExportCalendar_false) expresses the nice-to-have functional requirement

TABLE 3 Variability model for “Educational” SoS

UC_{Id}	UC description
UC_1	UC create student(u:administrativeStaff; s:student) pre not createdStudentAtAOSandLMS(s) and logged(u) post createdStudentAtAOSandLMS(s)
UC_2	UC create teacher (u:administrativeStaff; t:teacher) pre not createdTeacherAtAOSandLMS(t) and logged(u) post createdTeacherAtAOSandLMS(t)
UC_3	UC create course(u:administrativeStaff; c:course; cat:category) VPSettingCategory(SettingCategory_true, SettingCategory_false) pre not createdCourseAtAOSandLMS(c) and logged(u) and not created(cat) post createdCourseAtAOSandLMS(c) post created(cat) VPSettingCategory(true) post set(c, cat) VPSettingCategory(true)
UC_4	UC create class(u:administrativeStaff; c:course; t:teacher; cl:class) pre createdTeacherAtAOSandLMS(t) and logged(u) and createdCourseAtAOSandLMS(c) post createdClassAtAOSandLMS(cl, t)
UC_5	UC enroll student into a class(s:student; cl:class) pre createdStudentAtAOSandLMS(s) and createdClassAtAOSandLMS(cl, t) and logged(s) post enrolledStudentAtAOSandLMS(s, cl)
UC_6	UC create lesson (t:teacher; cl:class; l:lesson; d:date; m:material) VPSynchronousLesson(SynchronousLesson_true, SynchronousLesson_ false) pre createdClassAtAOSandLMS(cl, t) post createdLesson(t, cl, l) post scheduledLesson(t, l, d) VPSynchronousLesson(true) post sentMaterial(t, l, m)
UC_7	UC join lesson (s:student; cl:class; l:lesson; d:date; m:material) VPSynchronousLesson(SynchronousLesson_true, SynchronousLesson_ false) pre enrolledStudentAtAOSandLMS(s, cl) and createdLesson(t, cl, l) post lessonJoin(s, l, d) VPSynchronousLesson(true) post downloadMaterial(s, l, m)
UC_8	UC start chat (t:teacher; l:lesson; d: date; ch:chat) VPSynchronousLesson(true) pre scheduledLesson(t, l, d) post startedChat(t, l, ch)
UC_9	UC join chat (s:student; l:lesson; ch:chat) VPSynchronousLesson(true) pre lessonJoin(s, l, d) and startedChat(t, l, ch) post joinedChat(s, l, ch)
UC_{10}	UC set assignments (t:teacher; cl: class; a:assignment; l:lesson) VPSendNotification(SendNotification_true, SendNotification_ false) pre createdLesson(t, cl, l) post createdAssignment(t, a, l) post notificationSent(t, a) VPSendNotification(true)
UC_{11}	UC manage deadline (t:teacher; s: student; cl:class; a:assignment; d:deadline; l:lesson) pre createdAssignment (t, a, l) post deadlineSet(t, a, l, d) post deadlineNotificationSent(t, a, d) VPSendNotification(true) post seeAssigmentPersonalAgenda(s, a, cl, d) VPImport-ExportCalendar(true)
UC_{12}	UC create calendar event (t:teacher; cal:calendar; cl:class; e:event) VPImport-ExportCalendar(Import-ExportCalendar_true, Import-ExportCalendar_ false) pre createdClassAtAOSandLMS(cl, t) post created calendarEvent(cal, e) post exportedCourseCalendar(cal) VPImport-ExportCalendar(true)
UC_{13}	UC see calendar event (s:student; cl:class; cal:calendar; e:event) pre created calendarEvent(cal, e) post seenCalendarEvent(s, cal, e) post exportedCourseCalendar(cal) implies personalAgendaEventSeen(s, e, cal) VPImport-ExportCalendar(true)

of importing/exporting the course calendar. The variation attribute VPImport-ExportCalendar(true) allows indeed to select the SoS variant in which the teacher exports the calendar course (UC_{12} use case) and the student can see the calendar events (UC_{13} use case) and its assignments (UC_{11} use case) in his/her personnel agenda.

As showed in Table 3, for “Educational” SoS, the SoS engineer defined four nice-to-have functionalities expressed as variation points and associated variation point attributes. Table 4 (second column called $VPAttr$) summarises all the variation points attributes that could be present in an SoS variant of “Educational” SoS. Taking into account these variation point attributes, a set of SoS variants has been derived and considered by the SoS engineer for implementation.

For sake of simplicity, in order to derive the set of SoS variants for “Educational” SoS, we considered all the $k - combinations$ with $k \geqslant 0$ [27] of the 4 variation point attributes of Table 4 (second column), deriving a set of 16 possible different SoS variants, that were put forth to the SoS engineer for consideration. The application of different approaches for the derivation of the SoS variants, may have led to a different number of SoS variants to be considered.

5.4 | Desirability of “Educational” SoS variants

The set of the 16 SoS variants derived as explained in Section 5.3 were ranked according to their desirability. As explained in Section 4.4, the desirability metric was computed in terms of expected benefit and cost of each SoS variant.

TABLE 4 Variation point attributes of SoS variants in “Educational” SoS

VP	VPAttr	VPA_score
VPSettingCategory	SettingCategory_true	25
VPSynchronousLesson	SynchronousLesson_true	25
VPSendNotification	SendNotification_true	30
VPImport-ExportCalendar	Import-ExportCalendar_true	20

For computing the benefit of each SoS variant, we assigned a score (see last column called VPA_{score} of Table 4) to each variation point attribute (column VPAttr) so that their sum was 100[8]. The highest score was that of SendNotification_true variation point attribute (VPA_{score} equal to 30), this implies that having the SendNotification functionality in the implemented SoS variant is considered highly beneficial. The scores of SettingCategory_true and SynchronousLesson_true were the same (VPA_{score} equal to 25), this means that the SettingCategory and SynchronousLesson nice-to-have functionalities take the same importance in the desired SoS.

Then, we computed the benefit of each SoS variant of “Educational” SoS according to Equation 1 in the range [0,100]. For instance, let us consider two of the 16 SoS variants for “Educational” SoS, with identifier SoS_1 and SoS_2 and defined as:

$SoS_1 = \{SettingCategory\_true, SynchronousLesson\_true, SendNotification\_true, Import - ExportCalendar\_true\}$

$SoS_2 = \{SettingCategory\_true, SendNotification\_true, Import - ExportCalendar\_true\}$

we computed their benefit as:

$Benefit(SoS_1) = 25 + 25 + 30 + 20 = 100$

$Benefit(SoS_2) = 25 + 30 + 20 = 75$

The benefits of the 16 SoS variants under consideration for the “Educational” SoS ranged from a low value of 20 (reached by the SoS variant in which only the $Import-ExportCalendar\_true$ variation point attribute of Table 4 was present) to a high value of 100 (reached by an SoS variant in which all the variation point attributes of Table 4 were present, namely SoS_1 variant).

After computing the benefit of each SoS variant, we computed the expected cost of implementing each SoS variant in terms of the cost of the available CS instances able to cover the variation point attributes present in each SoS variant, according to Equation 3.

In particular, Table 5 shows for each CS type involved in the “Educational” SoS, the available CS instances identified during the CS recognition phase (see Section 5.2), along with their cost, estimated simply on a [1-10] scale. In particular, the available concrete instances of LMS were Moodle, Google Classroom and FullTeaching, whereas the available instances of the Calendar System were Google Calendar and Yahoo Calendar [9].

The first column of Table 5 shows the set of variation point attributes derived for “Educational” SoS that could be present in an SoS variant, whereas columns two and three show the subset of these variation point attributes that are present (marked as yes) into the same SoS variants with identifier SoS_1 and SoS_2 presented before, respectively.

Moreover, the table also shows the variation point attributes for “Educational” SoS that the concrete CS instances were able to cover (see rows from 5 to 9). Note that in order to compute the cost, the Import-ExportCalendar_true variation point attribute was split into the ExportCalendar_true and ImportCalendar_true variation point attributes corresponding to the functionalities that the LMS and the Calendar System must cover respectively in order to satisfy the Import-ExportCalendar_true variation point attribute.

[8]Note that for aim of simplicity the table does not show the VPA_{score} of the remaining variation point attributes of “Educational” SoS, namely SettingCategory_false, SynchronousLesson_false, SendNotification_false and Import-ExportCalendar_false, that has been set equal to zero.

[9]The AOS was fixed and instantiated by `RosarioSiS`, then we did not consider its cost in the computation of the SoS variant cost.

TABLE 5 CSs instances vs variation point attributes of “Educational” SoS

			LMS			Calendar System	
			Moodle C=5	Google Classroom C=6	Full Teaching C=4	Google Calendar C=1	Yahoo Calendar C=2
	SoS1	SoS2					
SettingCategory_true	yes	yes	✓		✓		
SynchronousLesson_true	yes				✓		
SendNotification_true	yes	yes	✓	✓			
ExportCalendar_true	yes	yes	✓	✓			
ImportCalendar_true	yes	yes				✓	✓

Applying Algorithm 1, we selected: i) Moodle, Full Teaching and Google Calendar for implementing SoS_1 with a total cost equal to 10; ii) Moodle and Google Calendar for realizing SoS_2 with a total cost equal to 6. Analogously, we computed the cost of the remaining SoS variants for “Educational” SoS.

Finally, for each SoS variant, the desirability metrics were computed according to Equation 4. The 6 highest desirability scores of the SoS variants for “Educational” SoS are showed in Table 6. SoS_2 results to be the selected SoS variant since it yields the highest desirability score, equal to 12.5. Moodle and Google Calendar are then chosen as the concrete implementations for LMS and CS respectively, for the development of SoS_2.

We will show in the next sections how to generate test cases for SoS_2 variant through model simulation and test objectives derivation.

TABLE 6 The top 6 desirability scores for “Educational” SoS

SoS Variant ID	SOS Variant Description	Desirability Score
SoS_2	{SettingCategory_true, SendNotification_true, Import-ExportCalendar_true }	12.5
SoS_4	{SettingCategory_true, SendNotification_true}	11
SoS_1	{SettingCategory_true, SynchronousLesson_true, SendNotification_true, Import-ExportCalendar_true }	10
SoS_3	{SettingCategory_true, SynchronousLesson_true, SendNotification_true}	8.88
SoS_7	{SendNotification_true, Import-ExportCalendar_true}	8.33
SoS_5	{SynchronousLesson_true, SendNotification_true, Import-ExportCalendar_true }	7.5

5.5 | “Educational” SoS simulation model

In this phase, a set of enhanced use cases for the SoS_2 variant (the one with the highest desirability) is derived parsing the variation points of the enhanced use case model of “Educational” SoS presented in Table 3 and applying the algorithm of requirements extraction presented in Nebut et al.'s approach[12,13]. The obtained set of enhanced use cases for SoS_2 includes all the use cases of Table 3 except for UC_8 (start chat) and UC_9 (join chat) since they refer to the variation point attribute SynchronousLesson_true that is not present in SoS_2 variant.

The SoS engineer proceeds to derive a set of test cases to assess SoS_2 functional requirements. As explained in Section 4.5, by simulating the use cases of SoS_2 variant, a UCTS is built. Specifically, this UCTS has been derived applying the algorithm by Nebut et al.[12], which successively tries to apply each instantiated use case of SoS_2 from the current state until all the reachable states have been explored. An instantiated use case is applied when its precondition is true with respect to the set of true predicates of the current state. The application of an instantiated use case allows to create an edge from the current state to a state of the system in which the postcondition is true. The theoretical maximum size of this UCTS could be high if all the states are evaluated. The UCTS maximum size depends on the number of instantiated predicates (p)[12], and is computed as $maxsizeUCTS = 2^{nip}$ where $nip = p \times maxistances^{maxparam}$, where maxistances is the maximum number of instances, and maxparam is the maximum number of parameters per predicate. In the use case model of SoS_2 there are 20 predicates, then assuming to have one instance for each predicate, the maximum size of UCTS for SoS_2 is equal to 2^{20} = 1.048.576 states. In practice, the actual size of the UCTS could be smaller since many potential states could not be reachable. An extract

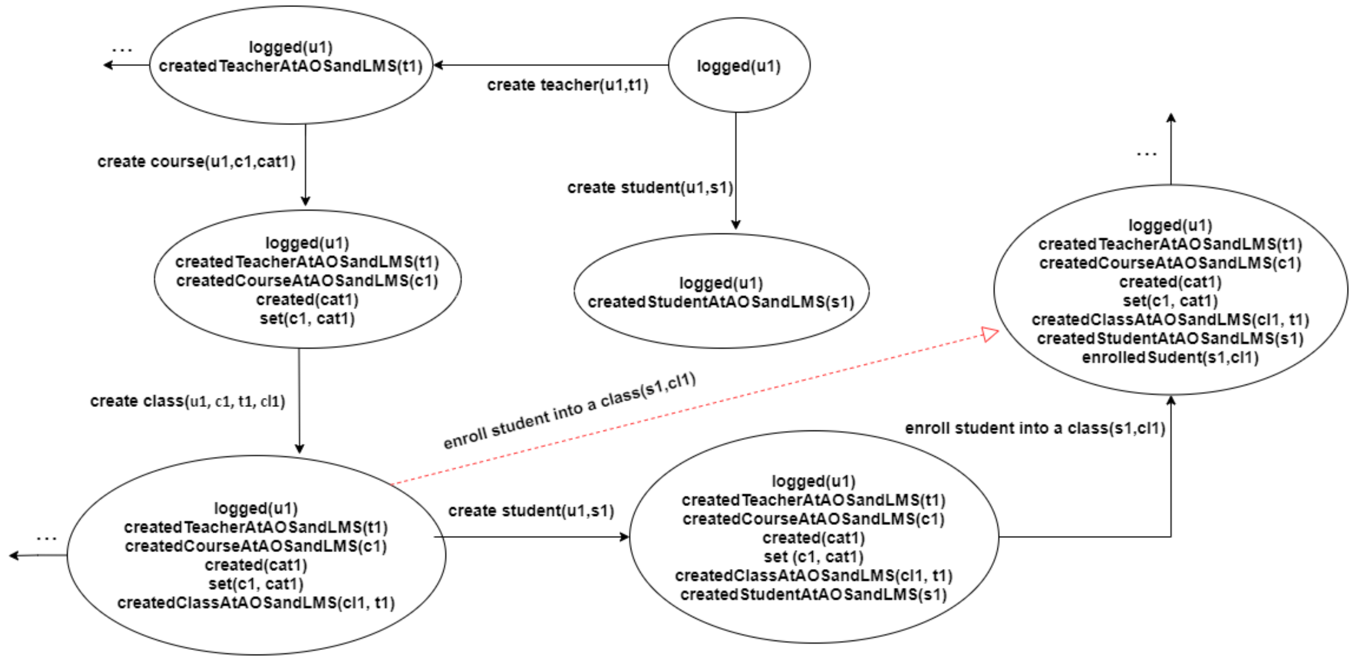
logged(u1)
createdTeacherAtAOSandLMS(t1)
create teacher(u1,t1)
logged(u1)
create course(u1,c1,cat1)
create student(u1,s1)
logged(u1)
createdTeacherAtAOSandLMS(t1)
createdCourseAtAOSandLMS(c1)
created(cat1)
set(c1, cat1)
logged(u1)
createdStudentAtAOSandLMS(s1)
logged(u1)
createdTeacherAtAOSandLMS(t1)
createdCourseAtAOSandLMS(c1)
created(cat1)
set(c1, cat1)
createdClassAtAOSandLMS(cl1, t1)
createdStudentAtAOSandLMS(s1)
enrolledSudent(s1,cl1)
create class(u1, c1, t1, cl1)
enroll student into a class(s1,cl1)
enroll student into a class(s1,cl1)
logged(u1)
createdTeacherAtAOSandLMS(t1)
createdCourseAtAOSandLMS(c1)
created(cat1)
set(c1, cat1)
createdClassAtAOSandLMS(cl1, t1)
create student(u1,s1)
logged(u1)
createdTeacherAtAOSandLMS(t1)
createdCourseAtAOSandLMS(c1)
created(cat1)
set (c1, cat1)
createdClassAtAOSandLMS(cl1, t1)
createdStudentAtAOSandLMS(s1)

FIGURE 5 Extract of the UCTS for “Educational” SoS_2 variant.

of the UCTS obtained for the SoS_2 variant of “Educational” SoS is shown in Figure 5. This UCTS is used for deriving the test objectives and the test cases as we explain in the following Section 5.6.

5.6 | Test cases derivation for “Educational” SoS

In this section we show some test objectives, test scenarios and executable test cases derived for the “Educational” SoS.

5.6.1 | Test objectives

As explained in Subsection 4.6, we can derive the functional test objectives for the SoS_2 variant, as the sequences of instantiated use cases of the simulation model of SoS_2 variant so to satisfy the APT criterion. Considering for instance to exercise the instantiated use case “enroll student into a class (s1, cl1)” (this is an instantiation of UC_5 of Table 3) whose precondition is “createdStudentAtAOSandLMS(s1) and createdClassAtAOSandLMS(cl1, t1) and logged(u1)”, we explore the values of the predicates that make the precondition true. In this example, the precondition is true only if all of its parts are true. Thus, referring to the simulation model of Figure 5, we can derive the test objective TO_5 corresponding to the path that from the initial state leads to a valid application of this use case, namely a state that satisfies its precondition, and then we can apply this instantiated use case. Since the initial state is logged (u1), a possible path resulting by the application of “enroll student into a class (s1, cl1)” is [create teacher(u1,t1), create course(u1,c1,cat1), create class(u1,c1,t1,cl1), create student(u1,s1), enroll student into a class(s1,cl1)].

To apply robustness testing, we derive from the simulation model the robustness test objectives, obtained by the invalid application of an instantiated use case. Considering for instance the same instantiated use case “enroll student into a class (s1, cl1)”, a robustness test objective leads to the invalid application of this use case, namely to an assessment of the system that violates its precondition. We represented this path by the red dotted arrows in the UCTS in Figure 5. Thus, if we consider the path [create teacher(u1,t1), create course(u1,c1,cat1), create class(u1,c1,t1,cl1), enroll student into a class(s1,cl1)], it leads to the execution of the “enroll student into a class (s1, cl1)” instantiated use case, but violating its precondition “createdStudentAtAOSandLMS(s1)”, so the system is expected to handle it in a specific way, for example, by issuing an error message.

Table 7 summarizes the test objective examples generated by applying the APT criterion and the robustness test criterion to the extract of the UTCS of the SoS_2 variant presented in Figure 5. Specifically, the first column represents the exercised use case, the second and third columns represent respectively the identifier and the corresponding path of the derived test objective, whereas last column specifies if it represents a functional or robustness test objective.

TABLE 7 Examples of test objectives for "Educational" SoS

UC_{Id}	TO_{Id}	TO_{Path}	Functional (F)/ Robustness(R)
UC_1	TO_1	[logged(u1), create student(u1,s1)]	F
UC_2	TO_2	[logged(u1), create teacher(u1,t1)]	F
UC_3	TO_3	[logged(u1), create teacher(u1,t1), create course(u1,c1,cat1)]	F
UC_4	TO_4	[logged(u1), create teacher(u11,t1), create course(u1,c1,cat1), create class(u1,c1,t1,cl1)]	F
UC_5	TO_5	[logged(u1), create teacher(u11,t1), create course(u1,c1,cat1), create class(u1,c1,t1,cl1), createStudent(u1, s1), enroll student into a class (s1, cl1)]	F
UC_5	TO_6	[logged(u1), create teacher(u11,t1), create course(u1,c1,cat1), create class(u1,c1,t1,cl1), enroll student into a class (s1,cl1)]	R
		....	

5.6.2 | Test scenarios generation

Following the strategy proposed by Nebut et al.[14,12], to fill the gap between the test objectives, which are derived at the requirements level, and the test cases, which are at the implementation level, we generate the application test scenarios. These test scenarios are derived from the test objectives by associating a scenario, expressed as a UML sequence diagram, to each instantiated use case contained in the test objective. The UML sequence diagrams can represent nominal or exceptional scenarios. The first represents the basic way to exercise a use case successfully; the latter represents a way to exercise a use case leading to a failure or error message. Sequence diagrams may contain more information than the use cases and, therefore, may express more detailed pre and post conditions than use case[14,12]. Referring to the "Educational" SoS, we show in Figure 6 the sequence diagram we derived for the nominal scenario of the instantiated use case "enroll student into a class (s1, cl1)". As shown in Table 3, this use case has "createdStudentAtAOSandLMS(s1) and createdClassAtAOSandLMS(cl1, t1) and logged(u1)" as precondition. We transform this precondition into Object Constraint Language (OCL)[10] restrictions that check if the student and the class exist in both AOS and LMS and if the admin user is logged. The "requestToEnroll" and "StudentSchedule" messages in Figure 6 correspond to the steps necessary to enroll a student in a class in the RosarioSiS system, which we had not considered during the analysis phase. The "enrolledStudentAtAOSandLMS(s1,cl1)" postcondition is also transformed into an OCL constraint that verifies that after performing the steps indicated in the diagram, the student is enrolled in the class and he/she has access to that class in the LMS. Sequence diagrams deal with testing at the system level. Therefore, taking into account the context of the SoS, sequence diagrams consider the actors and all constituent systems involved in the use case. In this example, the systems involved in this nominal scenario are RosarioSiS and Moodle.

To obtain the application test scenarios, we transformed test objectives, both for functional and robustness testing, into sequence diagrams, replacing all the instantiated use cases belonging to the test objective by their test scenarios. Considering the "Educational" SoS, to generate the test scenario corresponding to the nominal test objective TO_5 presented in Table 7, we replace the instantiated use cases UC_2, UC_3, UC_4, UC_1, and UC_5, by their nominal scenarios as shown in Figure 7. Analogously, to derive the robustness test scenarios, we replace all the instantiated use cases belonging to the robustness test objective by their test scenarios, but the last instantiated use case is always replaced by an exceptional scenario. We show in Figure 8 the test scenario for TO_6 robustness test objective of Table 7.

5.6.3 | Executable test cases derivation

From the derived sequence diagram, we can use tools that explore the system's behavior specifications to generate test cases automatically. For each scenario, a test method is created corresponding to a test case. The code of these tests comprises successive calls to the methods corresponding to the sequence diagrams messages with the actual parameters and assertions derived from the OCL contracts. For instance, Listings 1 and 2, correspond respectively, to the scenarios presented in Figures 7 and 8.

The test scenarios may be incomplete, depending on the sequence diagrams that are derived from. Only when the sequence diagrams contain precisely the same messages as the methods invoked on the considered constituent systems we can use a test scenario as a test case without any adaptation. Further, it is not possible to know in advance details of the GUI used in tools that simulate users' interaction in end-to-end tests. In this sense, we mark this step as semi-automatic, and we foresee that the tester must carry out the implementation of these interaction methods. Listings 3 shows the implementation of the *studentCanAccessClassInMoodle* method using the tools JUnit

[10]for Object Constraint Language, see https://www.omg.org/spec/OCL/2.4/PDF.

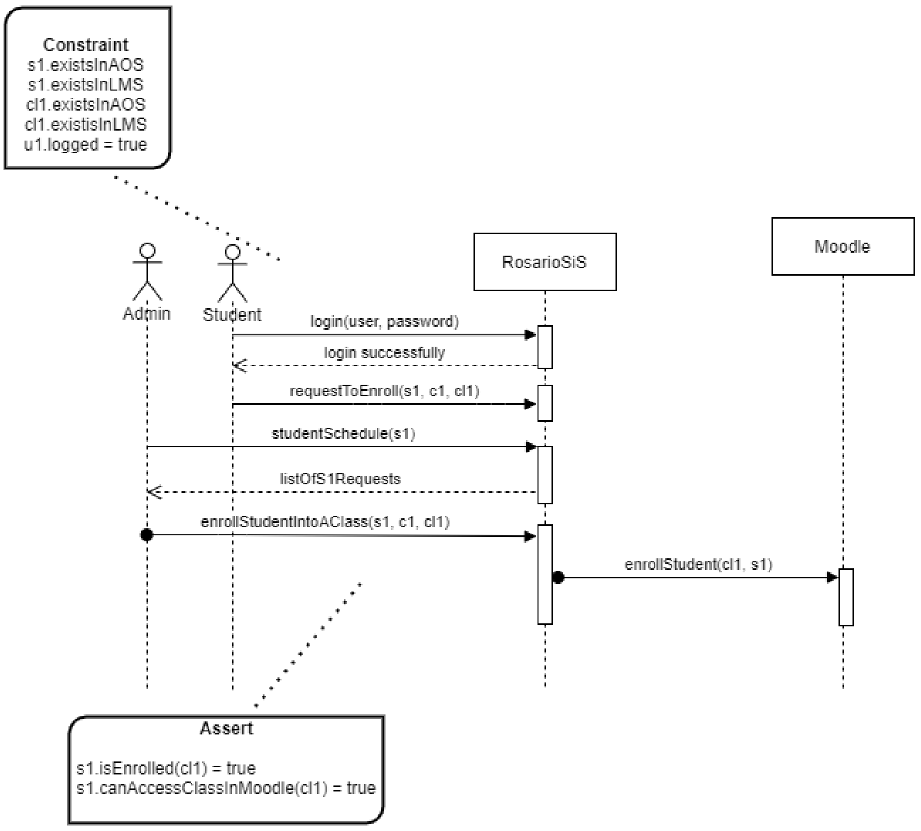

FIGURE 6 Sequence diagram for nominal scenario for use case UC_5.

Listing 1: Automated test case for test scenario for use case UC_5.

```
    private User u1;

    @Test
    public void enrollStudentIntoAClassTest() {
        User t1 = this.createTeacher();
        Subject c1 = this.addCourse();
        Class cl1 = this.createClass(t1, c1);
        User s1 = this.createStudent();
        Assertions.assertTrue(login(s1));
        this.requestToEnroll(s1, cl1);
        this.enrollStudentIntoAClass(s1, cl1);
        Assertions.assertTrue(this.studentIsEnrolled(s1, cl1));
        Assertions.assertTrue(this.studentCanAccessClassInMoodle(s1, cl1));
    }
```

[11] and Selenium [12]. In this method, which is responsible for checking if a student can access a class on Moodle, it is possible to see that details of the existing system's user interface were used to implement the test case.

[11] https://junit.org
[12] https://www.selenium.dev/

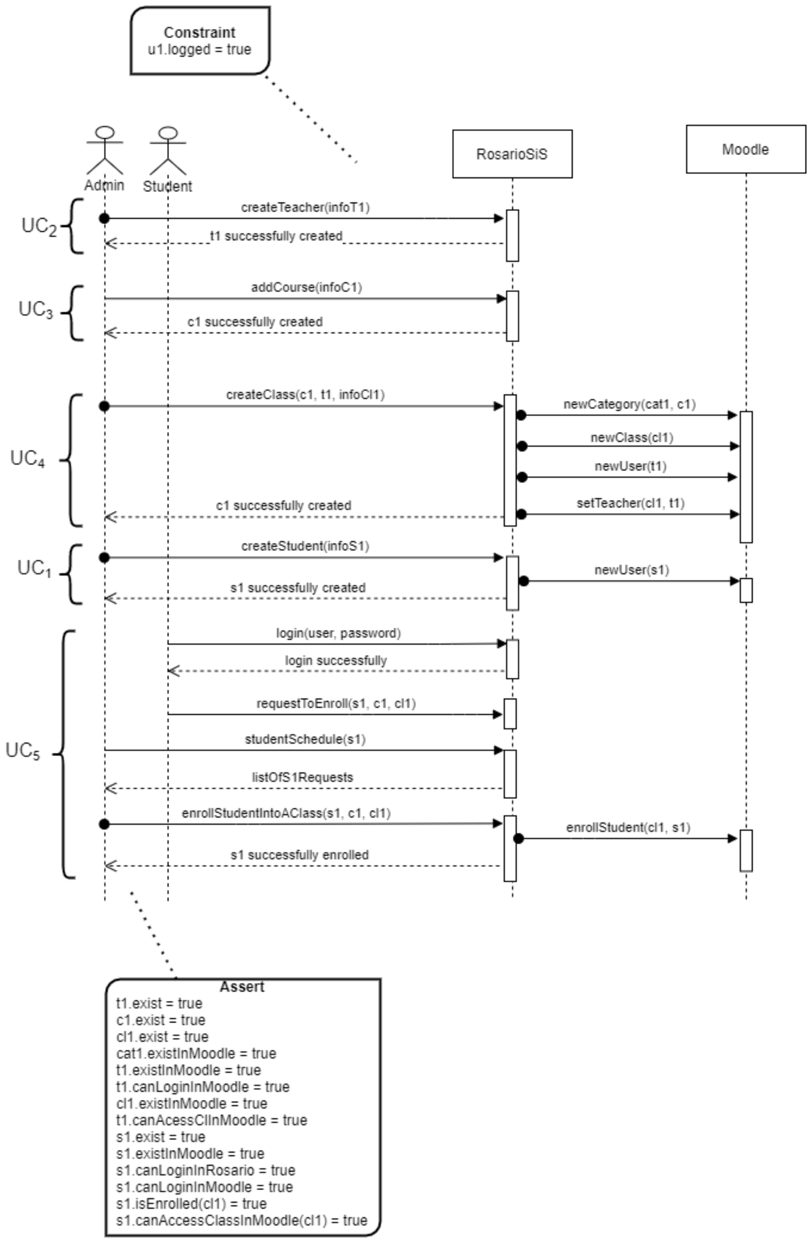

FIGURE 7 Nominal test scenario for use case UC_5 (test objective TO_5).

6 | FOCUS GROUP

The focus group is an empirical method in software engineering[28] designed to obtain the expert opinions of a group of practitioners or researchers about a defined area of interest. It consists of a carefully planned discussion within a limited number of participants, who are asked during a moderated group interview a predefined list of questions about the research focused.

The method is not suitable for testing hypotheses or obtaining quantitative assessments that are difficult to catch into a time-limited session[29], however it can provide a fast and cost-effective means to obtain an initial feedback on new concepts or ideas, by leveraging the experiences of the group members[30].

Our aim in conducting the focus group was, in fact, to receive a first evaluation of VANTESS by discussing the benefits and issues of the approach within a group of six academic experts. In the following sections, we first describe the main aspects of the adopted methodology and then provide an analysis of the obtained results.

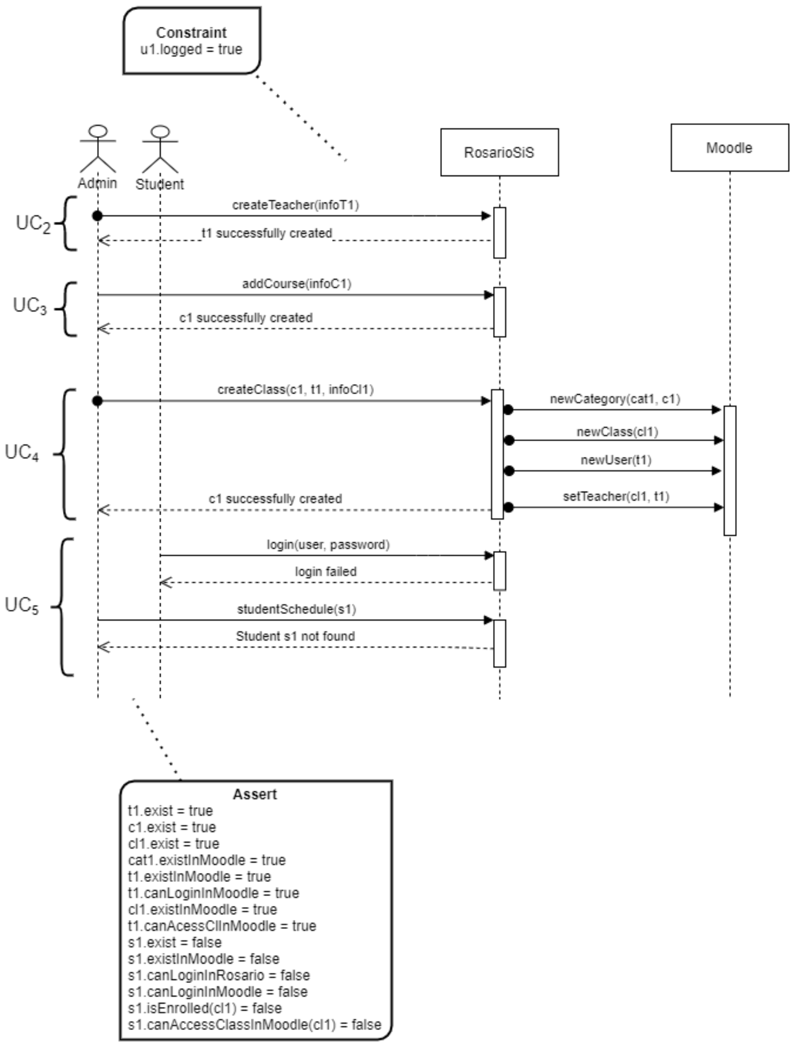

FIGURE 8 Exceptional test scenario for use case UC_5 (test objective TO_6).

6.1 | Methodology

In the focus group we addressed the first four phases of VANTESS, namely Define at abstract level the SoS overall mission, Make a recognition of available CSs, Model variability, and Assessing the desirability of SoS variants. These are the phases that best characterize the novelty of our approach, and that we wanted to discuss with the academic experts.

TABLE 8 Participants to the Focus Group and their expertise

	SoS	PLs & Variability	SW Architecture	SW Requirements & Modeling	SW Testing
$P1$	good knowledge	good knowledge	good knowledge	expert	familiar
$P2$	familiar	good knowledge	outsider	expert	familiar
$P3$	expert	familiar	expert	good knowledge	familiar
$P4$	familiar	good knowledge	good knowledge	good knowledge	familiar
$P5$	familiar	familiar	expert	expert	outsider
$P6$	expert	familiar	good knowledge	good knowledge	familiar

Listing 2: Automated test case for exceptional test scenario for use case UC_5.

```
    private User u1;

    @Test
    public void enrollStudentIntoAClassExceptionalTest() {
        User t1 = this.createTeacher();
        Subject c1 = this.addCourse();
        Class cl1 = this.createClass(t1, c1);
        User s1 = this.createNotRegisteredStudent();
        Assertions.assertFalse(login(s1));
        this.requestToEnroll(s1, cl1);
        Assertions.assertFalse(this.studentIsEnrolled(s1, cl1));
        Assertions.assertFalse(this.studentCanAccessClassInMoodle(s1, cl1));
    }
```

Listing 3: Example of real implementation with JUnit and Selenium.

```
    public boolean studentCanAccessClassInMoodle(User s, Class cl) {
        WebElement userMoodle = driverMoodle.findElement(By.id("username"));
        userMoodle.sendKeys(s.getUsername());
        WebElement passMoodle = driverMoodle.findElement(By.id("password"));
        passMoodle.sendKeys(s.getPassword());
        WebElement buttonLoginMoodle = driverMoodle.findElement(By.xpath("//*[@id=\"loginbtn\"]"));
        buttonLoginMoodle.click();
        driverMoodle.get(urlMoodle+"user/profile.php?id="+s.getId());
        List<WebElement> courses= driverMoodle.findElements(By.xpath("//*[@id=\"
            yui_3_17_2_1_1616083860288_22\"]/section[1]/ul"));
        return courses.size() > 0;
    }
```

Specifically, the focus group involved six academic experts recruited from different universities and research institutes in computer science around the world, and selected by convenience sampling. They form a homogeneous group with expertise as well as relevant publications in the following subjects: systems of systems, product lines & variability, software architecture, software requirements & modeling, software testing. Table 8 summarises the level of expertise self-declared from the anonymized participants for each research subject. Specifically, the level of expertise declared by the participants is: outsider (the participant judges him/her self as not expert at all), familiar (the participant knows the topic and follows its literature, but he/she has not worked much on it), good knowledge (the participant worked on it although may not be up-to-date with the latest literature), expert (the participant worked for years in the topic as a researcher and knows well the state of the art).

The focus group session lasted for two hours, was conducted remotely and recorded for later transcription. It involved the six academic experts as in Table 8 and the three paper authors, one of them taking the role of moderator. As shown, most of the participants indicated to be experts or to have good knowledge in at least two of the considered research subjects. About software testing, all the participants declared to have a low level of expertise (familiar or outsider). However, software testing is addressed in the remaining phases of VANTESS that are not object of this focus group.

After a preliminary step in which the moderator introduced the rules and objectives of the study, the focus group session was structured into two main parts: in the first part, one of the paper's authors (different from the moderator) did a short presentation about some background notions concerning the SoS architectures, the SoS design, the selection of constituent systems and the SoS variability, and then a group discussion was raised guided by the research questions (see below). The goal of this first part was to give a shared background and identify whether the variability provided by the available CSs is a concern when designing SoSs. In the second part, the other author who was not moderating presented the first four steps of VANTESS, again followed by the group discussion guided by research questions; in this second part we wanted to assess whether VANTESS can be useful to deal with variability when designing SoSs and identify strengths and weaknesses when using VANTESS to guide future work. In this direction we also sought advise concerning how the approach could be validated.

In particular, in the first part of our session we investigated the following Research Questions(RQs):

- RQ1.1 Which approaches do you know for the selection of constituent systems when conceiving an SoS?
- RQ1.2 How important/challenging is it to manage variability during SoS modeling/design?
  - RQ1.2a: Which approaches do you know that could be used or adapted?
  - RQ1.2b: As far as you know, could PL approaches be used/adapted?

In the second part, we investigated the following RQs:

- RQ2.1: Please comment usefulness/issues of VANTESS with regard to handling SoS variability.
- RQ2.2: Which aspects could be improved and how?
- RQ2.3: How could the approach be validated?

All along with the session, the role of the moderator was to keep the focus on the RQs, to encourage all the participants to be frank and express their opinions without concerns, and to limit the discussion's sidetracks.

6.2 | Results

All answers provided during the focus group have been recorded, transcribed and anonymized[13]. Then, we analyzed these data using the Thematic Analysis Template (TAT) method[31], which is a particular form of thematic analysis[32] used in organizational and management research[33] and other disciplines[34] for examining textual data including focus group transcripts. TAT has already been used in software engineering for analysis and reporting of focus group results[35].

In TAT, data analysis is based on the development of an initial coding template representing an initial set of the themes that are studied; then, according to the analysis of further data, this template can be revised and refined. The approach is very flexible about the style and the format of the template, as well as the development of the initial coding template. The themes of the initial template can be derived carrying out a preliminary coding of the data, as in most thematic approaches, or on the basis of a subset of data, or they can be a set of themes a priori identified and considered relevant according to the author's experience[34].

In our case, we created an initial template whose themes were set in advance of coding, according to the defined research questions (which in turn had been formulated on the basis of the aspects of VANTESS that we wanted to investigate in the focus group). Table 9 shows this initial template.

TABLE 9 Initial TAT template with themes and discussion aims

ID	Theme	Discussion
1	Variability management	Reveals issues and challenges of managing variability during SoS design
2	Constituent systems selection	Reveals existing approaches that could be adopted for selection of CSs when conceiving an SoS
3	Analogy of SoS with SPL	Reveals commonalities in managing variability between SoS and SPL
4	VANTESS approach	Reveals usefulness and issues of VANTESS
5	VANTESS validation	Reveals how VANTESS could be validated

The TAT was independently carried out by two authors manually (i.e., without using specialized software), who analyzed the transcribed data and provided two preliminary versions for the TAT final template. These two versions were very similar, with small disagreements regarding the chosen themes. Then, we held a meeting of all the authors in which we compared and aligned the respective TAT versions.

[13]The anonymized data are available at the VANTESS GitHub repository, available at https://github.com/edufysos/vantess

Table 10 presents the final set of themes, articulated into sub-themes. In the final template two new themes (SoS evolution and SoS cost) have been introduced with respect to the themes of the initial template. The results of the TAT analysis are presented below, grouped by themes and sub-themes.

TABLE 10 Final TAT template with the themes and sub-themes identified after data analysis

ID	Theme	Sub-theme
1	Variability management	1.1 SoS variability modeling
		1.2 Adapting existing approaches from other domains
2	Constituent systems selection	2.1 Reusing existing CS
		2.2 Adapting existing approaches from other domains
3	Analogy of SoS with SPL	3.1 Features selection
		3.2 Feature models
		3.3 SPL versioning
4	SoS evolution	4.1 Dynamic SoS architecture
		4.2 Evolutionary architecture simulation
5	VANTESS approach	5.1 Benefits
		5.2 Issues
6	VANTESS validation	6.1 What to do
		6.2 What not to do
7	SoS variant cost	7.1 Cost computation
		7.2 Refinement steps

1) Variability management

1.1) SoS variability modeling. All the participants agreed that modeling SoS variability is very challenging. They identified the following main challenges: i) the model size could be very large when considering all the possible SoS variants; ii) the granularity of the CSs knowledge is very important for defining variability models. One of the participants declared that a problem is indeed “Abstraction and completeness of the description that you have for your system, because depending on what’s your model, then you can take some decisions”. iii) Moreover, even though a complete knowledge of CS functionalities could be achieved, their dynamic evolution would make it extremely difficult to model all their variable functionalities; finally, iv) for some participants also the interaction with CSs stakeholders represents an issue for SoS variability modeling.

1.2) Adapting existing approaches from other domains. The participants concluded that no specific approaches for modeling SoS variability exist, and they suggested adapting solutions coming from other domains. Specifically, the suggested approaches were related to: i) feature models synthesis, based on requirements or textual inputs; ii) feature engineering; iii) simulation of dynamic architectures; and iv) SPL management (the latter domain repeatedly emerged during the discussion and is developed as a separate theme below).

2) Constituent systems selection

2.1) Reusing existing CS (from public repositories). The participants agreed that reusing existing CSs available into public repositories without applying a specific selection strategy represents the common practice when developing an SoS. One of the participants declared indeed “If I’m thinking as a developer the first thing I will do is to give a look at existing systems in GitHub, that might be reused for building my own system of systems’’.

2.2) Adapting existing approaches from other domains. As for SoS variability management, also for CSs selection, participants declared not knowing any specific approach and suggested adapting solutions coming from other domains. Some participants perceived an analogy of SoSs with service-based systems, namely choreography and orchestration, for what concerns the composition at design time of the SoS. When CSs selection happens at runtime, the same participants perceived an analogy of SoSs with self-adaptive systems where the continuous changes and updates of CSs are triggered by changes of the context and can be analyzed by a monitoring system. Other participants suggested to investigate solutions related to search-based software engineering and, more in general, to optimization problems, based for instance on genetic algorithms, to face the problem of CSs selection. One participant declared "The idea is using optimization to reduce the number of systems and maximize the number of functionalities you need to draw". Finally, solutions around goal modeling and especially the KAOS approach were considered suitable methods in the SoS context, not for the selection of the optimal SoS configuration but for the modeling of the different goals of the system.

3) Analogy of SoS with SPL

3.1) Feature selection. The focus group confirmed a strong analogy between SoS and SPL, which was, in fact, the initial premise of this paper. Many participants revealed similarities among SoS and SPL for what concerns both variability management and CSs selection. A participant declared "It seems that there are some analogies with software product lines where you define the number of features of your systems, and then, essentially when you want to create a product, you select the features, or fix the variance that you might want to have for your final system".

3.2) Feature models. Feature models have been considered suitable models for selecting features, obtaining system configurations and expressing variability. The participants suggested feature models synthesis based on textual requirements or natural language inputs to collect features description and try to generate feature models of what is available in the market for a specific purpose.

3.3) SPL versioning. The SoS representation has been considered more complex with respect to that of SPL, due to the dynamic SoS configuration. In particular, it has been identified an analogy with SPL versioning, namely SoS configuration could be compared to multiple snapshots of the same software product line, but with different conditions.

Theme 4) SoS evolution

4.1) Dynamic SoS architecture. A challenging aspect that emerged during the focus group is the dynamic SoS architecture, namely the SoS could change over time as the CSs are replaced. A participant declared: "SoS in general has dynamic or evolutionary architectures. This happens because the CSs can come and go, can join and leave the SoS at runtime. And so this is a challenge to manage variability". The dynamic nature of CSs during the time and the evolution of their interfaces represent an issue to be considered for both variability management and CSs selection. Also, the SoS mission could deviate from the initial specification, and as a consequence, the set of CSs and functionalities that should be integrated to accomplish that mission at a certain time of the SoS operation should change. Finally, another challenging aspect to be considered is the evolution of SoS requirements. A participant declared: "Fixing the requirements at the beginning and not being able to update them during the life of the system, I think it's a risky assumption because today I think there is no system where requirements don't change".

4.2) Evolutionary architecture simulation. Some participants remarked that to address the dynamic SoS evolution in modeling variability and CSs selection, it is needed that all the necessary functions or capabilities expressed in the variability model be present in the SoS at runtime, independently of CSs changes. To face this problem, one of the participants suggested adapting the existing approaches for simulation of evolutionary SoS architectures so to predict the different SoS configurations that could occur at runtime[36].

5) VANTESS approach

5.1) Benefits. Participants found that VANTESS targets an interesting topic and addresses many (or even too many) challenging aspects related to the different phases of the development lifecycle. One of the participants declared "I just want to join $P5$ in saying that you have a lot of courage in proposing an entire method and not focus on just a tiny little bit. So, it's good to have research like that". Another participant added "We needed to have some tool like this". They recognized that the main novelty of the approach is variability management in SoS. More than one participant appreciated its ease of use. One of the participants declared "You are doing some selection of the optimal variant in a way that is quite easy to understand and simple, and, that's fine''. Two of the participants agreed that a strong point of the presented approach is that it is general and independent from the deployment environment of the SoS as well as from the addressed domain. The approach has indeed been considered useful for managing CSs that could belong to different domains such as robotics or smart cities. In particular, robots have been considered good instances of independent systems that could cooperate into an SoS.

5.2) Issues. Some of the issues raised were related to the presentation rather than the approach itself. Two of the participants complained that VANTESS presentation during the focus group was not focused on showing the strong points of the approach. In particular, they advised to stress two main aspects: i) the novelty and distinguishing aspects of the proposed variability management approach with respect to existing solutions adopted in search-based software engineering, SPL, project management, and requirements management areas; ii) which are the strong points of VANTESS that are the object of the validation, namely if the validation aims to cover the entire approach or a part of it, and finally if it aims to show the efficiency in terms of time for deriving the best SoS variant. The focus discussion eventually allowed them to better grasp these aspects. One participant indeed declared: “After the discussion is more clear to me, what is the focus and what could be the benefits''. Concerning more specifically the proposed methodology, the participants raised the following main limitations of VANTESS: the approach does not consider non-functional requirements such as compatibility or interoperability among CSs; the dynamic evolution of CSs and SoS requirements are not addressed; dependencies among SoS variants are not considered.

6) VANTESS validation

6.1) What to do. The participants recognized the difficulty of validating a new approach or method as VANTESS. They declared: “It's not the tool that you run an experiment and now you're done. It's really more challenging”. One participant suggested performing VANTESS validation by means of a controlled experiment with a set of users who could be not confident with the development of SoSs. The idea was that the users could apply VANTESS in a constrained environment, then choose pieces of software from a public repository and combine them to accomplish the goals specified in the requirements. Then, the usefulness and user experience of the approach could be evaluated by questionnaires or interviews. Other approaches have been suggested to assess the usefulness of VANTESS: i) simulation models that should allow to exercise different SoS variants and predict their functional behavior; ii) methods, similar to mutation testing, able to detect some intentionally introduced changes in the system, by comparing the obtained results with the expected ones. Independently of the type of evaluation, it has been remarked that “It's important to perform validation on more than one system, and across domains”. Specifically, the participants suggested that VANTESS could be validated in the following domains: i) robotics in which different functionalities like vision, object recognition, or navigation are implemented as systems that are completely independent of the robot in which they are going to be run; ii) smart cities, using real data from available repositories for making SoS simulation; and finally, iii) civil protection and communication for risk awareness.

6.2) What not to do. It has been remarked that it would be unfair to compare VANTESS application with traditional development approaches where no variability management is applied. About the controlled experiment suggested to make the validation of VANTESS, some discussion arose about the opportunity of using students (who could not be expert of computer science disciplines) to assess if VANTESS represents a good educational approach. The participants also proposed to perform a survey with expert architects who have the knowledge and experience closer to reality to estimate the cost of a configuration over another.

Theme 7) SoS variant cost

7.1) Cost computation. An interesting discussion was raised about the cost computation of an SoS variant. More than one participant agreed that, concerning the cost computation, an issue of VANTESS is that it only associates the cost to a single CS without evaluating the cost of compatibility or interoperability among more CSs. A participant said: “If you have a very strong choice, but it's not compatible with all the other, and you cannot integrate with the rest of the components, then you are stuck and the cost is maximum”. An agreement emerged that another factor that should be considered into SoS variant cost computation is the dependencies among CSs, hence the cost of a CS should also consider the cost of the other CSs it depends on.

7.2) Refinement steps. Participants agreed that how to measure integration and interoperability costs among CSs still represents an open challenge. Estimating this cost by considering all the possibilities of integration of the available CSs could likely cause a combinatorial explosion and make the approach unusable. Participants suggested to introduce in VANTESS a cost computation refinement step, by which after an SoS variant has been selected, the integration costs associated to the CSs belonging to that SoS variant are estimated. Some more sophisticated metrics are needed to estimate such costs. Moreover, simulation or experimentation could help to know if some changes in the CSs could affect the cost of integration.

Threats to validity

As for other qualitative studies, potential threats to validity of the focus group results should be considered. The researchers who organized the focus group session are the same authors and developers of VANTESS, this could have been introduced biases in the planning, execution and results analysis of the focus group. To mitigate this risk, we followed an objective and rigorous methodology[28] along with all the phases of the focus group: in no way neither the moderator nor the other authors intervened in the discussion beyond the planned presentations, not to influence the orientation of the group. In addition, analysis of data has been performed independently by two authors of the paper and results have been compared and aligned. Moreover, we make available the data transcription to allow other researchers to evaluate the validity of the results. Another threat of the focus group session is related to the identification of the participants. This is an intrinsic threat of all focus group sessions, we tried to mitigate this issue by recruiting academic experts from different universities and research institutes. Moreover, they represent a well complemented group, whose expertise covers all the research subjects considered relevant for our study. Other possible threats are related to well-known focus group weaknesses[30]: i) business relationships between participants could have influenced their opinion; ii) group dynamics or communication styles among the participants could have influenced the results; iii) short discussion time could have forbidden participants to deeply discuss and understand more complex aspects. These are unavoidable threats of these studies. In our focus group session, business relationships or potential conflicts of interests among participants and with the authors were not present. To mitigate threats ii) and iii), the moderator clearly explained at the beginning of the session the rules and objectives of the focus group, and she balanced the discussions trying to involve less active participants.

7 | RELATED WORK

The works related to our proposal span over different research directions that are:

Designing SoS

Proper methodologies supporting the design of SoSs are still missing. On the other hand, methods and tools used to design monolithic and large-scale systems do not appear to work for SoSs[3]. Lana et al.[5] provide a comprehensive literature review of formal and semiformal languages that have been used for modeling SoS requirements grouped from model-based to property-oriented ones. According to this literature review, UML (specifically activity diagram, class diagram, and sequence diagram) and its variants such as SysML are largely adopted for defining SoS requirements.

Model-based approaches represent a promising direction for the analysis and development of SoS[6]. Different types of models are used in the different stages of SoS design, from mission model used by the application domain expert to the architecture model used by the system architect. Cherfa et al.[6] offer a procedure for explicitly specifying the SoS end-to-end mission and generating the appropriate abstract architecture. New architecture description languages, such as SosADL[37] have been specifically conceived to formally describe abstract architectural emergent behaviors of SoSs to be further refined according to the availability of the constituent systems of the SoS[38]. Mori et al.[39] propose a SysML[40] profile for SoSs providing novel architectural concepts and language constructs able to support automated SoS modeling and analysis.

mKAOS[23], which we introduced already in Section3.3, is a pioneering language that supports the specification of missions and the definition of relationships between such missions and the other elements of the SoS. Other model-based solutions[38,6] exist that support both the analysis and the architecture specification of SoS. Specifically, Silva et al.[38] present a model-based refinement process to automatically derive architecture models represented in SosADL starting from mKAOS mission models, whereas Cherfa et al.[6] provide precise procedures for guiding the specification of the SoS mission and then the generation of the appropriate architecture, focusing on acknowledged SoSs.

The aim of our proposal is not to develop a new SoS modeling language, rather we aim at identifying a method for expressing the variability implicit in the possibility to connect different existing CSs. Indeed, we leverage mKAOS for the definition of the SoS mission and UML use cases for the definition of the SoS functional behavior. With regard to existing SoS design approaches, the novelty of VANTESS consists into a new method for expressing variability in the SoS functional requirements leveraging enhanced UML use cases models.

Testing SoS

Not only our SoS architecture specification supports the automatic derivation of different SoS variants, it also permits the semi-automated generation of test cases. SoS testing is challenging due to highly dynamic and evolutionary nature of SoSs. Distinct characteristics of SoS, including their operational and managerial independence, geographic distribution, distributed operational environment, evolutionary development, and emergent behavior greatly impact on their test and evaluation[41]. In a previous work[42] we discussed how and to what extent existing test techniques can be adapted to cope with SoS peculiarities. SoS testing deals with different activities at the SoS level, as well as at the CSs level. Although many challenges have been identified for SoS testing at different testing levels[11], not much research on SoS testing yet exists and only few approaches in the literature address testing solutions for SoS.

At integration testing level, the main problem is to test all possible interactions and execution flows that could arise when CSs join or leave the SoS at any time. To address this problem, Luna et al.[43] identify all relevant SoS entities and their interfaces and the flow of information between the CSs, and then propose combinatorial testing strategies for optimizing the testing and SoS evaluation. Liang and Rubin[44] use a randomization approach to design test cases for SoS that minimize the number of possible test cases during the integration of the CSs. Moreover, interoperability issues and dependencies among CSs should be considered during integration testing.

At system level, the high number of states and settings that an SoS can reach prevents the use of exhaustive or exploratory testing. Zapata et al.[45] define an approach similar to white-box testing to generate test cases for an SoS. They use a basic path testing approach, after modeling the CSs as the nodes of a control flow graph; however they do not carry out case studies and do not discuss how their approach can scale up in a scenario with many CSs.

Another issue in SoS testing is to deal with the huge number of changes that can occur in an SoS configuration. To this purpose, two different testing strategies could be adopted: i) on the one hand, regression testing should be used to ensure that each new SoS configuration does not cause any inappropriate emergent behavior. In this direction, Bertolino et al.[46] propose a conceptual framework to govern regression testing for collaborative and acknowledged SoS, based on the regression test objectives for each phase of the SoS and using an orchestration graph; ii) on the other hand, it could be useful to revise testing activities according to the SoS changes. In this direction, Hess and Valerdi[47] present the PATFrame framework that aims to predict when a test system needs to be adapted and tested using the information learned during the test process.

Rather than proposing yet another testing strategy for SoS, following our earlier reflections[11] we leverage existing approaches for test case derivation in the context of SPL[12] and show some test cases semi-automatically generated for the "Educational" SoS.

Handling variability for SoS designing and testing

Variability modeling has been deeply investigated in the context of SPL. It aims to express the commonalities and differences among the products within a family and represents one of the main research areas of variability management.

A recent tertiary study[19] provides an overview of existing variability models for SPL. According to such study[19] a large majority of solutions is based on feature modeling. There are also other mechanisms of expressing variability such as decision modeling or orthogonal models, structured natural languages, use cases with variability specialization or adaptation, other UML-based models or domain-specific languages.

In this paper, we define a UML use case-based SoS variability model. UML use cases are considered a valid approach for modeling functional requirements in SPL and several approaches in the literature focus on how to describe variability in use cases[18]. Several proposals aim to extend use case diagrams with additional modeling elements for expressing the different types of variability (options, alternative, optional alternatives)[48] or by adding UML tagged values expressing variation point attributes as in Nebut et al.'s approach[13].

Another common approach to model the variability of a software product line is represented by feature models that represent all the possible products of the SPL by defining relationships and constraints among features[49]. To deal with the combinatorial explosion of the number of derivable products in an SPL, existing testing solutions aim to test SPLs by generating test configurations that cover all the valid t-wise feature interactions of a feature model[50]. However, applying t-wise testing, the number of product configurations to be tested may still be too high in large models with respect to the budget allocated for SPL testing.

Then, by the analysis of feature models, automated approaches have been developed for making pruning and prioritization of the tests to be executed within a product line[51,52,53]. Such approaches adopt optimization functions as well as cost and value feature information to derive a sorted list of products to be tested[51], or use statistical testing techniques based on usage models expressed as discrete-time Markov chains for selecting products and generating test cases[52], or leverage search-based approaches for the generation of product configurations for t-wise testing for large SPLs and similarity heuristics for their prioritization[53]. A different goal is that of multimorphic testing, namely to derive a family of variants or morphs with the objective of assessing the quality of a test suite[54]. All these approaches try to optimize the tests execution for a set of products, sampling the product configurations to test or identifying relevant products to test or checking whether testing different variants of the same program yields significant differences in terms of tests results. The goal of our approach is different, it is not on deriving an effective testing strategy for a set of SoS variants but handling SoS variability at design-time to support the SoS architect in the derivation and selection of a set of SoS variants. Then, we show how to apply existing approaches for testing of SPLs for deriving a set of test cases for a specific SoS variant.

In this paper, we leverage the enhanced use case model proposed by Nebut et al.[13] and adapt it to the specific needs of modeling variability in SoS context.

Handling variability that occurs at model level represents a challenge of SPL testing due to the huge complexity associated to testing the large number of specific products that could be derived. Petry et al.[55] provide a roadmap of SPL model-based testing approaches and variability management techniques for the test cases derivation. The results of this roadmap outline that 68% of the analyzed studies treat variability both at domain and application level engineering and that the majority of these studies focus on handling variability at design-time. Two main basic strategies for test case derivation leveraging variability models can be identified in SPL, they are abstraction and parametrization[56]: the former consists in developing general domain test cases by abstracting from differences among possible variants and then refining these general test statements with respect to the chosen variant; in the latter, parameterized domain test cases are developed and application test cases are derived from binding the parameters. Specifically, Nebut et al.[14,12] adopt parameterized domain test cases: by combining parameters and constraints expressed at use case level, test patterns formulated as UML sequence diagrams are developed and then test cases for application testing are synthesized from such test patterns.

The main idea of this paper is to leverage variability modeling that has been deeply investigated in SPL literature for expressing variability in SoS functional behavior and managing this variability for SoS variants selection and testing purposes. We are only aware of few preliminary works that speculate about variability in SoS development[8,9,11]. Precisely, Klein et al.[8] investigate variation points in the common platforms used for SoS so that new SoS configurations can be easily derived. To analyze the benefits and costs of different variation implementations and time binding requirements, methods and practices from product line could be used. In particular, they plan to extend the cost and benefit estimation models for SPL with additional categories of costs and benefits specifically conceived to address the SoS scale and context[8]. Botterweck[9] deals with variability and evolution in SoS in a position paper discussing the relationships between the concepts of Systems of Systems Engineering and the Product Line Engineering. Specifically, he fosters a vision in which systems (in an SoS context) could be products of a product line and at the same time product lines could be seen as a technique to produce components in an SoS approach. This vision could be beneficial for SoS evolution and reconfiguration. Finally, in our previous workshop paper[11] we outline commonalities and differences between the SoS and SPL paradigms from the point of view of testing and investigate how existing SPL methods and tools addressing variability could be leveraged to fulfill the challenges of SoS testing.

All these early papers raise interesting questions and identify important challenges, but do not provide approaches or techniques to express and leverage variability modeling for designing and testing of SoS. To the best of our knowledge, we propose the first concrete solution to leverage variability modeling adopted in SPL for designing, selecting and testing the most desirable SoS variant.

8 | CONCLUSIVE DISCUSSION AND FUTURE WORK

We have introduced VANTESS, a new variability-aware approach for designing and testing of SoSs. More specifically we target directed and acknowledged SoSs, and we support an SoS engineer who aims at creating an SoS from the assembling of available CSs with other components that are adapted or ad-hoc developed.

In the early stage of SoS architecture definition, the choice of the concrete CSs that will form the SoS is based on an opportunistic approach that considers the different functionalities they offer as well as their contribution to fulfill the overall SoS mission weighted against benefits and costs. In this context, VANTESS supports the SoS engineer during the early design and testing phases by providing: i) a new method for expressing variability in the SoS functional requirements according to the functionalities offered by the available CSs. The proposed SoS variability model leverages enhanced use cases models and the variability concepts widely explored in the SPL domain; ii) a solution to master such variability and then the large number of possible SoS behaviors that could arise when considering the various functionalities that the different involved CSs could offer. In this direction, VANTESS provides a heuristic to assess the desirability of the different SoS variants guiding the SoS engineer in the selection of the CSs during the design stage; iii) for the selected SoS variant, VANTESS also allows to generate in a semi-automated way a test plan in terms of test objectives and then a set of executable test cases through model simulation and test scenarios derivation. These test cases will allow us to assess both the functional behavior of the SoS and its robustness.

We walked-through the application of all the phases of VANTESS to the “Educational” SoS. This application example, introduced in our previous work[24], has been here further developed according to the proposed approach and then fully implemented. We provide all the artifacts produced while applying the VANTESS approach at the VANTESS GitHub repository, available at https://github.com/edufysos/vantess, whereas the reference implementation of the selected constituent systems and their integration can be found from the EDUFYSoS repository at https://github.com/edufysos/edufysos.

To assess the approach, we conducted a focus group session with six experienced field participants. We transcribed the session and performed the qualitative analysis using TAT. The results obtained in the focus group allowed us to identify seven themes and fifteen sub-themes. Although participants were not aware of a specific approach to CSs selection and variability management, they pointed out that approaches from other domains could be adapted to deal with this problem. We highlight the SPL and KAOS approaches, which we consider in this work. Regarding the VANTESS approach, we found that the main pros are that it is a simple approach and considers the entire stage of the SoS development process, from its conception to testing. However, the cons are that it does not consider the evolutionary aspect of the SoS and it does not take into account the quality attributes.

Some limitations of the approach are related to costs computation. As said, the approach does not take into account the integration costs of CSs. These costs can be related to non-functional requirements, such as, for instance, compatibility costs between CSs, that would prevent their coexistence in the same SoS. Also, costs related to dependences on other systems that a given CS may have, for example, a CS that needs to authenticate itself using another system.

To assess the desirability of SoS variants, we leveraged and modified the CBAM method. With respect to CBAM, in VANTESS, the contribution score is very simple (1 or 0), it could be less refined than in CBAM but it can be automatically derived avoiding the manual assignment from the stakeholders. A limitation of VANTESS with respect to the original CBAM decision-making technique is the number of the involved stakeholders. In VANTESS, for aim of simplicity we consider only one stakeholder, but the approach could be extended to consider more than one stakeholder.

Another limitation of the benefit function adopted in VANTESS is that of considering the variation point attribute scores as single values into a defined range. More complex measures for the variation point attributes could be defined depending for instance on application scenarios or using utility level associated to the variation point attributes as done by Moore et al.[21].

While the SoS architect can check the consistency of each SoS variant's requirement using simulation of the use cases, the VANTESS approach does not yet allow to detect conflicting requirements. The management of emerging conflicting requirements in SoS is a challenging problem and existing approaches[57] in literature try to address it. We plan in the future to investigate state-of-the-art approaches for conflicting requirements detection in SPL and to extend VANTESS adapting them to the context of SoS design.

We conclude that the results obtained also open up opportunities for future research. VANTESS deals with modeling variability of the functional behavior of SoSs. In future, we plan to extend our approach in order to consider also the variability of quality attributes such as performance, interoperability, or security. For that, we can include a refinement step that calculates the CSs integration cost considering these quality attributes during the early design phase of the SoS. This extension of VANTESS will permit to take into account also these quality aspects to select the configuration of CSs that can better guarantee for instance a given performance or security level, or, on the other hand, that high interoperability costs, for instance, impede the realization of an SoS.

In this paper, we focused on handling variability of functional requirements of the SoS at the early design stage, i.e. over space. Our goal in the future is to investigate the challenging problem of how to manage the variability in time, i.e., when the SoS is running and the autonomous CSs implementing the desired functionalities may need to evolve to adapt to changes of the context. We plan to develop an extension of VANTESS able to support the SoS engineer in the iterative process of selecting for each new release of the SoS, the set of its functional requirements by considering the evolution of the existing CSs as well as the functionalities of new CSs that could be enrolled in the SoS to fulfill the new requirements.

We intend to conduct case studies to increase confidence in the validity and usefulness of VANTESS. For this, we may consider different SoS domains such as robotics and smart cities. Faults seeding techniques can be researched and applied in the context of SoS to analyze the effectiveness of the generated test cases.

ACKNOWLEDGMENTS

This paper has been partially supported by the Italian MIUR PRIN 2017 Project: SISMA (Contract 201752ENYB).

We would like to thank all the participants to the focus group for their insightful feedback.

References

1. Dahmann JS, Baldwin KJ. Understanding the current state of US defense systems of systems and the implications for systems engineering. In: Systems Conference. ; 2008: 1–7.

2. Nielsen CB, Larsen PG, Fitzgerald J, Woodcock J, Peleska J. Systems of systems engineering: basic concepts, model-based techniques, and research directions. ACM Computing Surveys (CSUR) 2015; 48(2): 1–41.
3. DeLaurentis DA. A taxonomy-based perspective for systems of systems design methods. In: IEEE international conference on Systems, Man and Cybernetics. ; 2005: 86–91.
4. Pohl K, Böckle G, Van Der Linden F. Software product line engineering: foundations, principles, and techniques. Springer . 2005.
5. Lana CA, Guessi M, Antonino PO, Rombach D, Nakagawa EY. A Systematic Identification of Formal and Semi-Formal Languages and Techniques for Software-Intensive Systems-of-Systems Requirements Modeling. IEEE Systems Journal 2018; 13(3): 2201–2212.
6. Cherfa I, Belloir N, Sadou S, Fleurquin R, Bennouar D. Systems of systems: From mission definition to architecture description. Systems Engineering 2019; 22(6): 437–454.
7. Lock R, Sommerville I. Modelling and analysis of socio-technical system of systems. In: 15th IEEE International Conference on Engineering of Complex Computer Systems. ; 2010: 224–232.
8. Klein J, Chastek G, Cohen S, Kazman R, McGregor J. An early look at defining variability requirements for system of systems platforms. In: Second IEEE International Workshop on Requirements Engineering for Systems, Services, and Systems-of-Systems (RESS). ; 2012: 30–33.
9. Botterweck G. Variability and evolution in systems of systems. In: 1st Workshop on Advances in Systems of Systems (AiSoS'13). ; 2013: 8-23.
10. Ncube C, Oberndorf P, Kark AW. Opportunistic software systems development: making systems from what's available. IEEE Software 2008; 25(6): 38–41.
11. Bertolino A, Lonetti F, Neves VO. Standing on the Shoulders of Software Product Line Research for Testing Systems of Systems. In: IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). ; 2020: 209–214.
12. Nebut C, Fleurey F, Le Traon Y, Jezequel JM. Automatic test generation: A use case driven approach. IEEE Transactions on Software Engineering 2006; 32(3): 140–155.
13. Nebut C, Le Traon Y, Jézéquel JM. System testing of product lines: From requirements to test cases. In: Springer. 2006 (pp. 447–477).
14. Nebut C, Pickin S, Le Traon Y, Jézéquel JM. Reusable test requirements for UML-modeled product lines. In: International Workshop on Requirements Engineering for Product Lines (REPL). ; 2002: 51–56.
15. Kazman R, Asundi J, Klein M. Quantifying the costs and benefits of architectural decisions. In: 23rd International Conference on Software Engineering (ICSE 2001). ; 2001: 297–306.
16. US Department of Defense . Systems Engineering Guide for Systems of Systems. Office of the Deputy Under Secretary of Defense for Acquisition and Technology, 2008.
17. Chen L, Babar MA. A systematic review of evaluation of variability management approaches in software product lines. Information and Software Technology 2011; 53(4): 344–362.
18. Santos IS, Andrade RMC, Neto PAS. How to describe SPL variabilities in textual use cases: A systematic mapping study. In: Eighth Brazilian Symposium on Software Components, Architectures and Reuse. ; 2014: 64–73.
19. Raatikainen M, Tiihonen J, Männistö T. Software product lines and variability modeling: A tertiary study. Journal of Systems and Software 2019; 149: 485–510.
20. Nebut C, Fleurey F, Le Traon Y, Jézéquel JM. Requirements by contracts allow automated system testing. In: 14th International Symposium on Software Reliability Engineering (ISSRE). ; 2003: 85–96.
21. Moore M, Kaman R, Klein M, Asundi J. Quantifying the value of architecture design decisions: lessons from the field. In: 25th International Conference on Software Engineering. ; 2003: 557–562.

22. Falessi D, Cantone G, Kazman R, Kruchten P. Decision-Making Techniques for Software Architecture Design: A Comparative Survey. ACM Comput. Surv. 2011; 43(4).

23. Silva E, Batista T, Oquendo F. A mission-oriented approach for designing system-of-systems. In: 10th System of Systems Engineering Conference (SoSE). ; 2015: 346–351.

24. Bertolino A, De Angelis G, Lonetti F, Neves VO, Olivero MA. EDUFYSoS: A factory of educational system of systems case studies. In: IEEE 15th International Conference of System of Systems Engineering (SoSE). ; 2020: 205–210.

25. Rothermel G, Untch RH, Chu C, Harrold MJ. Test case prioritization: An empirical study. In: IEEE International Conference on Software Maintenance (ICSM). ; 1999: 179–188.

26. Lane J. Cost model extensions to support systems engineering cost estimation for complex systems and systems of systems. In: 7th Annual Conference on Systems Engineering Research. ; 2009.

27. Karakashian S, Choueiry BY. Tree-Based Algorithms for Computing k-Combinations and k-Compositions. Tech. Rep. CSE Technical reports, University of Nebraska - Lincoln; 2010.

28. Kontio J, Bragge J, Lehtola L. The Focus Group Method as an Empirical Tool in Software Engineering. In: Shull F, Singer J, Sjøberg DIK., eds. Guide to Advanced Empirical Software EngineeringSpringer London. 2008 (pp. 93–116).

29. Edmunds H. The focus group research handbook. The Bottom Line 1999.

30. Kontio J, Lehtola L, Bragge J. Using the focus group method in software engineering: obtaining practitioner and user experiences. In: International Symposium on Empirical Software Engineering (ISESE'04). ; 2004: 271–280.

31. Cassell C, Symon G. Essential guide to qualitative methods in organizational research. Sage . 2004.

32. Guest G, MacQueen KM, Namey EE. Applied thematic analysis. Sage publications . 2011.

33. King N, Brooks J, Tabari S. Template analysis in business and management research. In: Ciesielska M, Jemielniak D., eds. Qualitative methodologies in organization studiesSpringer. 2018 (pp. 179–206).

34. Brooks J, McCluskey S, Turley E, King N. The utility of template analysis in qualitative psychology research. Qualitative research in psychology 2015; 12(2): 202–222.

35. Scanniello G, Romano S, Fucci D, Turhan B, Juristo N. Students' and professionals' perceptions of test-driven development: a focus group study. In: 31st Annual ACM Symposium on Applied Computing. ; 2016: 1422–1427.

36. Manzano W, Graciano Neto VV, Nakagawa EY. Dynamic-sos: An approach for the simulation of systems-of-systems dynamic architectures. The Computer Journal 2020; 63(5): 709–731.

37. Oquendo F. Formally describing the software architecture of systems-of-systems with SosADL. In: 11th System of Systems Engineering Conference (SoSE). ; 2016: 1–6.

38. Silva E, Cavalcante E, Batista T, Oquendo F. Bridging missions and architecture in software-intensive systems-of-systems. In: Int. Conf. on Engineering of Complex Computer Systems. ; 2016: 201–206.

39. Mori M, Ceccarelli A, Lollini P, Frömel B, Brancati F, Bondavalli A. Systems-of-systems modeling using a comprehensive viewpoint-based SysML profile. Journal of Software: Evolution and Process 2018; 30(3): e1878.

40. OMG . Systems modeling language (SYSML) specification, version 1.3 (June 2012). http://www.omg.org/spec/SysML/1.3/PDF; .

41. Dahmann J, Lane JA, Rebovich G, Lowry R. Systems of systems test and evaluation challenges. In: 5th International Conference on System of Systems Engineering. ; 2010: 1–6.

42. Neves VO, Bertolino A, De Angelis G, Garcés L. Do We Need New Strategies for Testing Systems-of-Systems?. In: IEEE/ACM 6th International Workshop on Software Engineering for Systems-of-Systems (SESoS). ; 2018: 29–32.

43. Luna S, Lopes A, Tao HYS, Zapata F, Pineda R. Integration, verification, validation, test, and evaluation (IVVT&E) framework for system of systems (SoS). Procedia Computer Science 2013; 20: 298–305.

44. Liang Q, Rubin SH. Randomization for testing systems of systems. In: IEEE International Conference on Information Reuse & Integration. ; 2009: 110–114.

45. Zapata F, Akundi A, Pineda R, Smith E. Basis path analysis for testing complex system of systems. Procedia Computer Science 2013; 20: 256–261.

46. Bertolino A, De Angelis G, Lonetti F. Governing regression testing in systems of systems. In: IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). ; 2019: 144–148.

47. Hess JT, Valerdi R. Test and evaluation of a SoS using a prescriptive and adaptive testing framework. In: 5th International Conference on System of Systems Engineering. ; 2010: 1-6.

48. Maßen v. dT, Lichter H. Modeling variability by UML use case diagrams. In: International Workshop on Requirements Engineering for product lines. ; 2002: 19–25.

49. Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS. Feature-oriented domain analysis (FODA) feasibility study. tech. rep., Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst; 1990.

50. Perrouin G, Oster S, Sen S, Klein J, Baudry B, Le Traon Y. Pairwise testing for software product lines: comparison of two approaches. Software Quality Journal 2012; 20(3): 605–643.

51. Galindo JA, Turner H, Benavides D, White J. Testing variability-intensive systems using automated analysis: an application to android. Software Quality Journal 2016; 24(2): 365–405.

52. Devroey X, Perrouin G, Cordy M, et al. Statistical prioritization for software product line testing: an experience report. Software & Systems Modeling 2017; 16(1): 153–171.

53. Henard C, Papadakis M, Perrouin G, Klein J, Heymans P, Le Traon Y. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. IEEE Transactions on Software Engineering 2014; 40(7): 650–670.

54. Temple P, Acher M, Jézéquel JM. Empirical Assessment of Multimorphic Testing. IEEE Transactions on Software Engineering 2021; 47(7): 1511-1527.

55. Petry KL, OliveiraJr E, Zorzo AF. Model-based testing of software product lines: Mapping study and research roadmap. Journal of Systems and Software 2020; 167: 110608.

56. Kamsties E, Pohl K, Reis S, Reuys A. Testing variabilities in use case models. In: International Workshop on Software Product-Family Engineering. ; 2003: 6–18.

57. Viana T, Zisman A, Bandara AK. Identifying conflicting requirements in systems of systems. In: IEEE 25th International Requirements Engineering Conference (RE). ; 2017: 436–441.

58. Mikkonen T, Taivalsaari A. Software reuse in the era of opportunistic design. IEEE Software 2019; 36(3): 105–111.

59. Temple P, Acher M, Jézéquel JM. Poster: Multimorphic Testing. In: IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion). ; 2018: 432–433.

60. DeLaurentis DA, Crossley WA, Mane M. Taxonomy to guide systems-of-systems decision-making in air transportation problems. Journal of Aircraft 2011; 48(3): 760–770.

61. Uday P, Marais K. Designing resilient systems-of-systems: A survey of metrics, methods, and challenges. Systems Engineering 2015; 18(5): 491–510.

62. Axelsson J, Fröberg J, Eriksson P. Architecting systems-of-systems and their constituents: A case study applying Industry 4.0 in the construction domain. Systems Engineering 2019; 22(6): 455–470.

63. Mori M, Ceccarelli A, Lollini P, Bondavalli A, Frömel B. A holistic viewpoint-based SysML profile to design systems-of-systems. In: IEEE 17th International Symposium on High Assurance Systems Engineering (HASE). ; 2016: 276–283.

How to cite this article: F. Lonetti, V. Neves, A. Bertolino, (2021), Designing and Testing Systems of Systems: From Variability Models to Test Cases passing through Desirability Assessment, JSEP, 2021;00:1–6.