Using Metamodels to Improve Model-Based Testing of Service Orchestrations

Lucas Leal
University of Campinas
Campinas, Brazil
ra163140@students.ic.unicamp.br

Leonardo Montecchi
University of Campinas
Campinas, Brazil
leonardo@ic.unicamp.br

Andrea Ceccarelli
University of Florence
Florence, Italy
andrea.ceccarelli@unifi.it

Eliane Martins
University of Campinas
Campinas, Brazil
eliane@ic.unicamp.br

***Abstract*—Online model-based testing is one of the most suitable techniques to assess the proper behavior of service orchestrations. However, the diverse panorama in terms of modeling languages and test case generation tools is a limitation to widespread adoption. We advocate that the application of Model-Driven Engineering principles as meta-modeling and model transformation can cope with this problem, improving the interoperability of artifacts in the test case generation process, thus bringing benefits in case of agile development processes, where system and technology evolution is frequent. In this paper, we present our contribution to this idea, introducing i) a reference metamodel, which stores the business process behavior and the information to generate input models for testing tools, and ii) transformations from orchestration languages towards testing tools. The proposed approach is implemented in a testing framework and evaluated on a case study where multiple orchestrations are expressed in two languages. Also, the paper presents how test cases are appropriately generated and successfully executed, starting from an orchestration model as a consequence of successful transformations.**

***Index Terms*—Model-Driven Engineering, SOA, Metamodeling, Model-Based Testing.**

I. Introduction

The Service-Oriented Architecture (SOA, [1]) paradigm was designed to permit loose coupling interactions among independent computational entities, called services, which are usually controlled by different owners and hosted in different locations [2]. Like microservices, SOA and its derivatives are the chosen solution to implement many of the systems that society depends on. These services support applications on the internet of things (IoT), cyber-physical systems (CPS), and Systems of Systems (SoS) [3], [4].

The SOA paradigm has two main composition patterns: Choreography and Orchestration [5]. Their difference is the presence of a component responsible for managing the interaction of the other components of the system. Orchestrations rely on the "orchestrator", which is usually controlled by the SOA composition owner, and it is responsible for the coordination of the service providers (proprietary and third-party) that contribute to the system objectives. Choreographies are protocols (or contracts) among service providers that contribute to the system's goals [5]. This paper focuses on service orchestrations, although concepts are extensible to other forms of SOA compositions.

In orchestrations, services are integrated blindly to their role in the compositions, which makes their replacement easier. However, orchestrations depend on the proper behavior of its constituent services: behavior changes and unexpected evolution in services may jeopardize the whole composition operation. These factors, combined with the dynamic binding of components, which postpones the composition's behavior evaluation, forces the validation process to be performed at the integration and runtime phases [6].

The popularization of continuous integration and testing caused by the adoption of agile software development processes and DevOps practices helped reducing the time between system releases [7]. The trend is that software testing now occurs parallel to software development and integration. However, the use of test scripts and automated test suites does not solve all the problems regarding system validation. There is a need for methods to perform efficient full life cycle testing and validation of the end-to-end business process, integrating capabilities, and different system components [8] [9].

In order to overcome some of the challenges regarding the validation process, the software industry invested in test automation techniques, and especially we consider in this paper Model-Based Testing (MBT, [10]). *Runtime Model-based testing* can ensure that the services, interfaces, messages, and business processes are behaving as expected [6]. Nevertheless, when considering services and SOA applications, i) the different MBT tools used to automate the testing process, ii) the service diversity and their descriptions, iii) the frequent need for system tests, iv) requirement changes, and v) the unpredictable evolution and availability of services, make the testing process not efficient and troublesome to be maintained [10].

In this paper, we advocate that the introduction of Model-Driven Engineering (MDE, [11]) techniques of model transformation and artifact generation can be applied to overcome the first and second identified challenges regarding SOA testing. Such approach facilitates the creation and execution of test suites for SOA applications, even when multiple testing tools or different SOA descriptions and service languages are

This work has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No 823788 "ADVANCE". This work has received funding from the São Paulo Research Foundation (FAPESP) with grant #2017/21773-9. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

considered.

This paper's main contribution is to discuss the possible applications of MDE for the improvement of the runtime MBT process on SOA applications, taking into consideration the current software development paradigm. The paper also presents two metamodels, three transformations to guarantee interoperability with different orchestration languages and testing tools, and a running implementation based in an existing framework [12]. The proposed approach was evaluated on a case study with orchestrations expressed in two different languages, which evaluated the model transformations and if the reference metamodel is capable of bridging different SOA descriptions and MBT techniques.

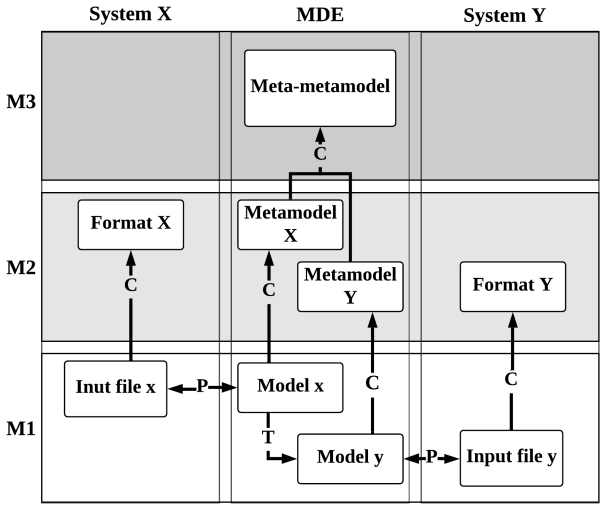
This paper is structured as follows. Section II introduces the basics. Section III explains the research questions and the methodology to devise the proposed solution. Sections IV and V contain the technical details of the proposed solution, presenting the metamodel and the transformations to generate artifacts from it. Section VI discusses the experimental results. Section VII discusses related works. Finally, Section VIII concludes the paper, including a discussion of our future works.

II. Fundamentals

A. Model-Driven Engineering

Model-Driven Engineering (MDE, [13]) relies on model abstractions, which represent information about a specific domain. The MDE paradigm is explained by dividing it by conceptualization levels and organizing it in implementation levels. Conceptualization levels are organized in application-level (M1), application-domain (M2), and meta-level (M3). We illustrate the division and organization of the MDE paradigm through Figure 1, which is also useful to explain the transformation methodology that we will use in the rest of the paper. The figure presents an example of a model transformation process between system X and system Y, divided into conceptualizations levels from bottom-up, and application from left to right [14].

Relevant core concepts are the concepts of model, metamodel, and meta-metamodel. A *model* is an abstraction of a system, and it may represent an existing system or just a possible design (e.g., the UML description of a software). A *metamodel* is also a model, and it defines structures and rules to which models based on it should conform (e.g., the UML metamodel [13]). A *meta-metamodel* defines the modeling-language that is used to describe metamodels [14] (e.g., the Meta-Object Facility [13], which is used to define the UML metamodel, among others). As exhibited in Figure 1, the M1 layer focuses on model definitions, transformation mechanisms, script and code generation. M2 focuses on the definitions of the modeling languages (metamodels) and their transformation rules. M3 declares the meta-metamodel, to which the metamodels, models, and transformations should conform [14].



C - Conformance relationship
P - Projection (text-to-model and model-to-text)
T - Model-to-Model transformation

Fig. 1. Conceptualization levels in MDE.

B. Model-Based Testing

Model-based testing (MBT) relies on models that reflect the behavior of the system under test (SUT) and of the environment, to automate the test-case generation processes. It is a variant of black-box testing; therefore, the test cases are generated without knowledge of the SUT source code. The tests target the SUT interface, asserting outputs accordingly to given inputs. Generated test cases are then utilized to evaluate the SUT [15].

The typical MBT process consists in i) generation of abstract test cases from the SUT behavior model, ii) implementation of concrete test cases from the abstract ones, and iii) execution of the test cases on the SUT. Customizing this process depends on the testers' requirements, such as the need to test individual features of the SUT (partial system models), stress one specific SUT behavior (test case selection criteria), or perform test case generation before or during test execution. This last aspect discriminates between offline or online model-based testing [32].

In this paper, since we are addressing the issues of testing dynamic SOA applications at runtime, online MBT has an advantage over offline MBT. Online MBT can cope with most of SOA's short deployment cycles and the unpredictable evolution/behavior of its components without storing pre-generated test cases.

C. The SAMBA Framework

SAMBA (Self-Adaptive Model-Based online testing for dynamic SOAs, [12]) is an online testing framework designed to perform regression tests on SOA orchestrations. It uses the information available in orchestration files to extract the business process of the orchestration and to generate a model, which is then used in the test case generation process.

SAMBA realizes the following functionalities:

1) updates the files describing the monitored orchestrations when requested or when changes in the orchestrations are detected;
2) extracts testing-oriented models from the orchestration's description files;
3) generates test cases from the testing-oriented models;
4) executes tests based on the generated test cases;
5) generates test reports.

Figure 2 presents the SAMBA components and their organization. SAMBA is composed of four main components: Service Assemble Monitor (SAM), Model Generator (MG), Model-based Online Test case generator (MOT), and Test Service (TS). The components are organized as MAPE-K [16] stages and have distinct functions. SAM is responsible for monitoring the evolution of the target SOA application, and it informs the MG which orchestration changed. MG analyzes the evolution of the target SOA application and generates updated models. MOT plans the test cases and updates the test report, and TS executes the runtime tests [12].

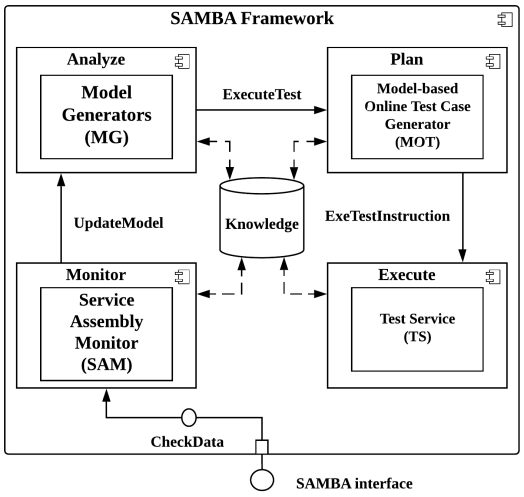


Fig. 2. SAMBA components and MAPE-K control loop [12].

The most noteworthy for the scope of this paper are the MG and the MOT. The MG generates the models of the monitored orchestrations. It manages copies of the orchestration *description files* and decides if model updates are necessary. The description files are used as a source of information for the model generation process.

The model generation is triggered whenever an application description file update is detected. When this happens, the orchestration description file is parsed, the information about the orchestration business process is extracted, and it is converted to a model format accepted by the model-based test case generation component.

Then, the MG requests to the MOT a test on the target application using the updated model. The MOT controls the test case generation process and the test report generation. It interacts with the MBT tool to generate test cases and with the TS to execute them. It verifies if all the necessary components are available, loads the updated model, sets initial parameters, defines stop conditions, and starts the test loop. While executing the test loop, the test instructions and the obtained test results are registered in a test report.

The current implementation of SAMBA has been developed *ad hoc* for orchestrations described with the Business Process Execution Language (BPEL, [17]), and it is compatible with only one specific MBT test case generation tool, namely GraphWalker [18]. In other words, the current model generation process in SAMBA restricts i) the language to describe SOA orchestrations, ii) the model-based testing tool used to generate test cases, and iii) their operational parameters.

III. Objective and Approach

A. Problem Definition

This research aims to investigate the benefits of applying the concepts of meta-modeling and model transformation from MDE to improve MBT processes, specifically in the context of runtime testing of service orchestrations. Testing at runtime is necessary whenever the system can not be fully tested before deployment, e.g., incremental development applied in agile development processes [19].

Generally, each MBT approach relies on specific modeling standards, syntax, test case generation, and test execution, often leading to incompatibilities between approaches [15]. MDE has many applications, among them improving interoperability through the use of platform-neutral abstractions (models) that must conform to a metamodel. Relying on a common, well-defined metamodel enables the generation of platform-specific artifacts according to requirements, thus improving the interoperability between tools/systems.

Therefore, MDE can simplify the artifact generation and compatibility for different MBT tools and frameworks. It offers the possibility to use model generation scripts for any MBT tool, at the single cost of defining transformation rules (set only once for each metamodel). Consequently, the effort for updates and extensions to the testing framework is reduced (for example, including new MBT approaches and tools), thus improving the testing process's automation level.

We list below the research questions that guide our research, and for which we will develop an answer in the rest of the paper:

1) *What is the typical information necessary to generate an input model to a standard MBT tool that targets SOA applications?*
2) *What are the challenges related to the transformation from different SOA description models to a generic model?*

B. Proposed Solution

We develop a reference metamodel for the testing of service orchestrations. The metamodel objective is to comply with the needs of MBT techniques. The metamodel stores the information required to generate input models for MBT tools. The metamodel's information structure enables the use of transformation rules, which allows the automation of the

model transformation from the system requirements and documentation. Therefore, the metamodel must be able to represent business processes from different description notations.

The first requirements of the metamodel were extracted from the target MBT tool. Then, by analyzing the characteristics of the subjects of the case study and their description models, it was possible to define the common elements in both notations that could be used as source information for the end goal. To exercise the metamodel, we create the following transformations:

i from the reference orchestration and business process languages, namely the BPEL [20] and the Business Process Modelling Notation (BPMN, [21]), towards the metamodel; and

ii from the metamodel towards the MBT tool Graphwalker.

The results of the transformations were applied to the SAMBA framework [12], which can perform runtime testing of orchestrations. Since SAMBA is an MBT framework, the model generation process is fundamental for the framework to achieve its objective.

Two components of the SAMBA framework were substituted: MG and MOT. The transformations in i) will replace the MG, and the transformations in ii) will replace the MOT.

This approach would end the need to generate a model generation script from each description model to each MBT tool, reducing the effort required during framework updates. The adoption of MDE techniques would also help to fulfill the requirements of MBT tools used in dynamic environments, improving the automation of the testing process, thus making it more suitable for agile development processes.

IV. A Metamodel for MBT of SOA Orchestrations

In this section, the metamodel used to perform model-based testing in SOA orchestrations is presented; from now on, it will be mentioned as *SOA Testing Metamodel* (STM). The metamodel can be downloaded from [22] and it is depicted in Figure 3.

As defined by Utting et al. [15], *model specification* in MBT is organized in three dimensions: model scope, model characteristic, and model paradigm. Based on this, we comment on STM characteristics and paradigm (the scope should be clear from the discussion above). The STM has the following *model characteristics*: i) it specifies the expected input-output behavior of the SUT; ii) it does not address the timing issues of the orchestration components; iii) it is deterministic since it is based on business processes; iv) it is discrete since it models the orchestration events and operations, v) it uses a transition-based notation as a modeling paradigm.

The metamodel is designed to represent SOA orchestrations, which are a type of SOA composition. Usually, the components of an SOA composition have their behavior well defined, are designed to be stateless, and are accessed by SOAP or REST communication protocols; these characteristics are represented in the metamodel.

For what concerns the *model paradigm*, the STM is based on Ecore, which is an implementation of the Meta-Object Facility (MOF, [24]). Ecore models can be designed using the well-known Eclipse Modeling Framework (EMF, [25]), a modeling framework capable of code generation from structured data models, with runtime support for models and a reflective API for their manipulation.

We finally discuss in detail Figure 3, which presents the STM in the Ecore notation. The *Model* class is the root class as the classes of the STM can be divided into behavioral and requirement classes. The model class is responsible for holding both behavioral and requirement objects. The behavioral classes are: *Step, StepOperation, OperationComposition, AtomicStepOperation, ParallelStepOperation, SequenceStepOperation*. All the remaining classes are requirements for the MBT process: *Generator, Service, Operation, Arguments, Oracle, OperationOracle, AssertionOracle, Parameters.*

The most important behavior class is the *Step*. The idea is to represent all the actions performed inside the business process as an atomic action. The structural information of the application is stored in the model steps relationship. This way, a model is a collection of steps that reference each other. The property nextStep of the Class Step, can have zero or many references to Step objects. A step has a unique ID attribute and is composed of one *StepOperation*.

The *StepOperation* is a supertype for the *AtomicStepOperation* and *OperationComposition*. The *AtomicStepOperation* may reference an actual service operation, which is convenient since different kinds of service providers, other than web-services, might perform actions inside the business process.

The *OperationComposition* is also a supertype for both *ParallelStepOperation* and *SequenceStepOperation* classes. These abstractions were made to allow the representation of concurrent and nested *StepOperations*, respectively. The STM can also describe concurrent *StepOperations*, which is useful only if the MBT test case generation tool can generate tests with concurrent operations.

The *Service* class holds the information of a service provider. Service objects have a unique ID, name, address, and communication port. Each service is composed of a collection of one or more *Operations*. Each *Operation* object has a name, a set of *Arguments* and *Outputs*, and can reference any number of *Oracles*. The *Service* and *Operation* classes are responsible for the interface between the STM's structural and behavioral classes.

The *Oracle* class holds the input-output information of the service operations consumed by the SOA orchestration. In the context of testing, oracles are artifacts for determining whether a test has passed or failed [26]. The *Oracle* class is a supertype for the *AssertionOracle* and *OperationOracle* classes. The *AssertionOracle* stores a logical operator, a set of input parameters, and an assertion value used to evaluate an operation. The *OperationOracle* holds two sets of input and output parameters for a given operation. The parameter class is designed to store the values used by the oracles to perform an operation execution and validation process, and it also references the argument to which it is bound. The *Argument* class, instead, has just the name and a data type

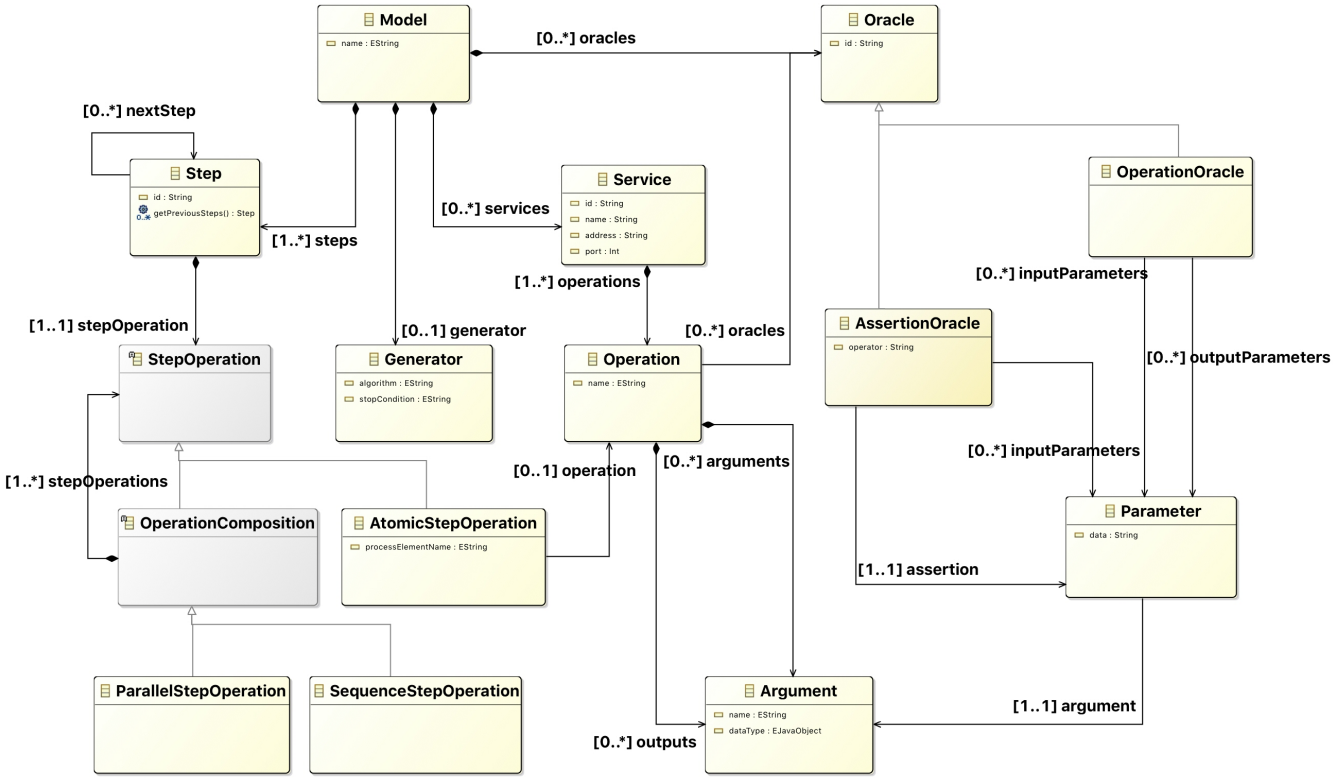


Fig. 3. SOA Testing Metamodel (STM) classes and relationships.

as attributes and it is strictly bound to an operation object.

V. Transformations

In this section we describe the metamodels, tools, and artifacts required and used by the transformations. First, Section V-A presents two metamodels, their classes, and relationships used in the transformation process. Second, Section V-B describes the required model-to-model (M2M) and model-to-text (M2T) transformation and the tools required to realize them. Third, in Section V-C the scripts for such M2M and M2T transformations are explained, jointly with their limitations. All transformations are available at [23].

A. Input metamodels: BPMN & LiteBPEL

We selected two starting metamodels, BPMN and BPEL. We used the implementation of the BPMN metamodel for Eclipse EMF available at [25].

The BPMN metamodel has an abstract superclass called *FlowElements* for all the subclasses used to describe a process flow (BPMN business process). The *FlowElements* are *Flownodes (Activities, Choreography Activities, Gateways, Events), Data Objects, Data Associations*, and *Sequence Flow* [21]. These classes are the main target of the transformation script that will be explained in Section V-C. A simplified version of the class diagram of the BPMN FlowElements subclasses is presented in Figure 4.

Concerning BPEL, the BPEL Ecore model available in the Eclipse Foundation repository was not stable. For this reason, we developed the LiteBPEL metamodel. LiteBPEL is a smaller

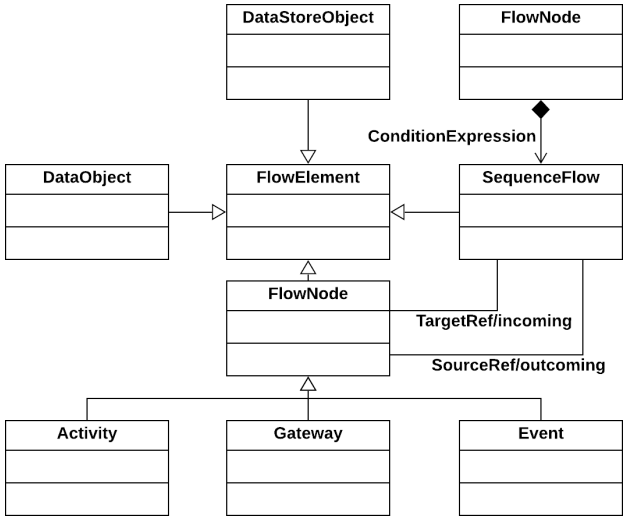


Fig. 4. BPMN FlowElements subclasses.

implementation of the BPEL metamodel and it conforms to the BPEL documentation [16]. The model holds only the information necessary for the subsequent transformation to the STM, which are: i) the sequence of the BPEL activities inside the process, ii) the variables consumed and shared in the process and activities, iii) the service providers that are consumed by the orchestration, and iv) their respective operations. Like the BPEL metamodel, LiteBPEL also has an abstract superclass representing the elements used to describe

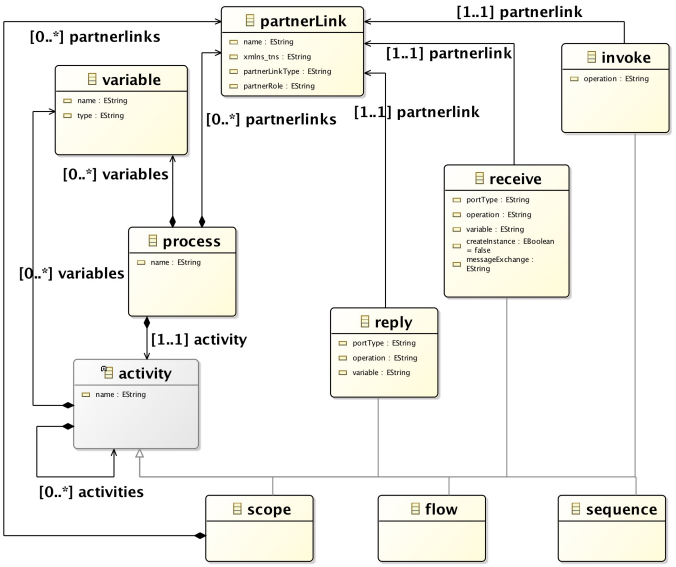


Fig. 5. The LiteBPEL metamodel.

a process (BPEL business process): the *Activity class*. Some of the Activity subclasses are: *Scope, Flow, Sequence, Invoke, Receive*, and *Reply*. The class *PartnerLink* holds information about the service providers used by the composition. The Variable class is a reference to a variable shared and used inside a process [27].

Just like original BPEL models, LiteBPEL models group activities under the classes *Scope, Flow, and Sequence*. The order of the objects describes the business process, which is the only information on the model that indicates the activities execution sequence. The *Scope class* indicates the use of a separate execution environment for the Activities that are inside it. The *Flow class* indicates that the Activities inside of it are executed in parallel. The *Sequence object* lists the activities inside on their proper execution order. Figure 5 shows the LiteBPEL metamodel.

B. The transformation workflow

The transformation workflow comprises two parts: a Model-to-Model (M2M) transformation and a Model-to-Text (M2T) transformation. The whole transformation process has been implemented and exercised in the case study, following the MDE approach, as displayed in Figure 1. From left to right, there are: two M2M transformations generating STM versions of the BPMN and BPEL orchestrations, and one M2T transformation generating Graphwalker input models. These are discussed in what follows.

1) M2M transformations: BPEL2STM and BPMN2STM: The two M2M transformations are from BPEL to STM and from BPMN to STM. Table I presents information about the two transformation codes produced for the case study. In both cases, the transformation process comprises the following steps.

- Description files of the orchestration according to a certain metamodel are retrieved, i.e., BPEL and BPMN orchestrations in the case study.

TABLE I
SUMMARY OF M2M TRANSFORMATIONS

Transformation	Rules	Lazy rules	Helpers	Lines of code
BPEL2STM	1	10	6	186
BPMN2STM	1	8	12	246

- The description files are projected into their equivalent models. In practice, this means that the XMI file representing the BPEL or BPMN orchestration has to be interpreted as a model.
- The models can be validated according to their respective metamodels. In fact, the description files must conform to their metamodel. This is an obvious requirement for the correctness of the model.
- Then, the models (BPMN and LiteBPEL models) are converted into new models that conform to the target metamodel (STM).

A set of rules performs model transformations. The transformation rules were implemented using the Atlas Transformation Language (ATL [28]). ATL is a project supported by the Eclipse Foundation. It is a model transformation language also designed to be an MDE toolkit, which provides M2M transformation solutions.

2) M2T Transformations: STM2Graphwalker: The SAMBA Framework relies on GraphWalker, an online MBT test case generation tool [18]. GraphWalker has built-in REST APIs with methods to load models, fetch data from the generated test case, restart, or abort the test case generation, get and set data from/to a model. GraphWalker requires a finite state machine (FSM) as an input model and a set of configurations for test case generation. The model can be described in JSON, while the configuration consists of the path generator (it determines the strategy to use when generating a path through a model) and a stop condition (it specifies when the path generation should stop).

The current implementation of the SAMBA framework is set to operate only with models described in JSON format. The GraphWalker JSON model format does not have an explicit metamodel, but on Graphwalker's wiki there is a textual description of how the models should be written [18].

The M2T transformations are performed to generate JSON files starting from the model in STM. A model in STM is used as an input file for an Acceleo [29] script developed for the case study. Acceleo is a model-based technology that includes design tools to aid in the code/text generation process; it automates the generation process from any data sources that conform to an available Eclipse Modeling Framework (EMF) metamodel. Table II presents more information about the *generate* script and its support scripts.

C. Details on the Transformation Algorithms

1) From LiteBPEL to STM Model: The first step in the transformation process is to generate STM *Services* from the BPMN *Partnerlinks*. Then STM *steps* are generated from the BPMN *Activities*. Each *Activity* is converted to an equivalent STM *StepOperation* subclass *(ParallelStepOperation, Se-*

TABLE II
SUMMARY OF THE STM2GRAPHWALKER TRANSFORMATION

File	Imports	Templates	Queries	Lines of code
generate	3	2	0	30
edgeGeneratorModule	1	6	0	60
generateHeaderModule	0	1	0	14
vertexGeneratorModule	1	1	0	13
GraphHelper	0	0	13	110

quenceStepOperation, AtomicStepOperation). Activities subclasses like *Invoke, Receive and Reply* are transformed into STM *AtomicStepOperations*. *Invoke* Activities with information about operations are converted to STM *Operation* objects and associated with its respective STM *Service*. The order of the Process' Activities is stored and later used to update the STM Step's *nextStep* property, linking them according to their original execution sequence.

2) From BPMN to STM Model: First, BPMN *Interfaces* are converted into STM *Services*. Then all the *Flownodes* and *Sequence Flows* are extracted from the main BPMN Process. *Flownode Gateways* are listed, and their information is stored. All the *FlowNodes* that are not *Gateways* or *Events* are converted to STM *Steps*. Finally, all the information from the *Sequence Flows* and *Gateways* is used to update the STM *Step*'s *nextStep* attribute, reestablishing the original execution sequence. Figure 6 illustrates the result of the transformation of an example BPMN model to an STM model.

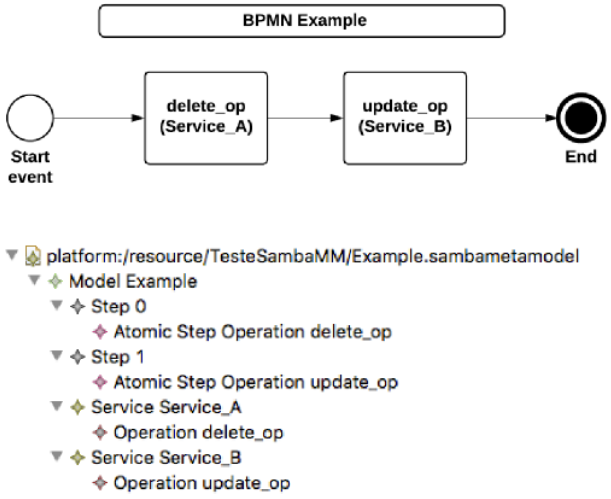


Fig. 6. BPMN to STM transformation example.

3) From STM Model to JSON: The STM has a collection of Steps. This class holds the information required to reconstruct the SOA business process. It is necessary to convert the business process information available in the STM to a simplification of a UML state machine to generate the Graphwalker JSON input file. We realize a graph where edges hold the information required to perform test operations: each edge's name refers to a *service operation* used in the orchestration, with the syntax *operation name@service*. The edges, which represent all the operations required to achieve

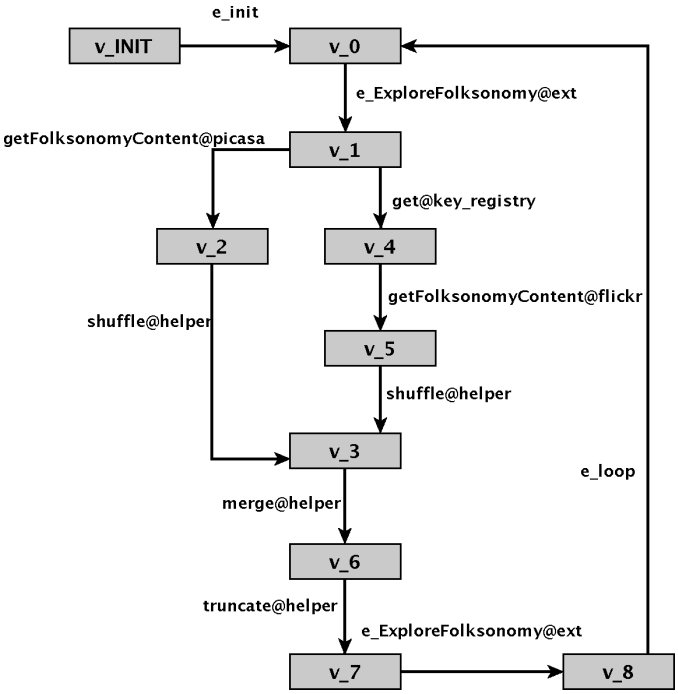


Fig. 7. Graphical representation of a sample transformation step.

the SOA business process, are generated from the information available in the class *Operation*. Vertices are created when necessary. Vertices link the edges and are required to represent the original business process properly.

In practice, the transformation begins with the creation of the first vertex and the first edge of the model. Next, STM *steps* are converted to clusters of edges and vertices, depending on the complexity of the structure defined by the *Step Object*. *Steps* generated from LiteBPEL models are usually more complex than *Steps* generated from BPMN models. The information of each STM *AtomicStepOperation* defines the name of the edge, which is used by the SAMBA Framework to trigger test executions. The STM *Step*'s *nextStep* parameters are used to connect the edges and vertices clusters.

A sample representation of model elements and their relations are graphically presented in Figure 7, for the orchestration model ImageScraper from [30] also used in the case study.

VI. CASE STUDIES WITH BPEL AND BPMN

A. Input Model and Tests Definition

The objective is to understand the relationship between the STM and the transformation scripts (M2M and M2T) to answer the research questions. The case study evaluates the M2M transformations designed to transform i) from orchestrations descriptions to the STM, and ii) from STM to a GraphWalker JSON model, which is the SAMBA Framework input model. The case study is composed of two groups of subjects: BPEL orchestrations from the jSeduite SOA [30] and BPMN2 business processes from jBPM examples [20].

jSeduite is an SOA application that organizes the information broadcast inside academic institutions. jSeduite is

composed of different BPEL orchestrations and it consumes services from thirty-one service providers. Three BPEL orchestrations were selected to test the transformation process: ImageScraper, HyperTimeTable, and FeedReader [30].

ImageScraper relies on six different operations and four WSs, which invoke additional WSs, including external services as Flickr and Picasa. HyperTimeTable relies on four different operations and two services (which invoke additional services that manage an online calendar). Finally, FeedReader uses two different operations and two services that, amongst the various things, invoke external services for feed reading.

Instead, jBPM enables users to automate business processes. The jBPM servers are extensible workflow engines written in Java that execute business processes using the BPMN 2.0 specification. Therefore, it can be used to host both SOA orchestrations and SOA choreographies since BPMN 2.0 is capable of describing such compositions [20]. The jBPM examples selected for the case study are Customers and Hiring processes, which are available on the jBPM 7.3 release and are composed respectively of 5 and 2 activities (atomic actions inside de business process).

The transformation scripts will be tested during the case study, and the resulting models will be checked for behavioral conformance with their source model. Also, to show that BPEL and BPMN transformation rules generate equivalent models, the BPEL ImageScraper was re-implemented in BPMN and was used to check the differences between BPMN and BPEL transformations.

B. Results

All orchestrations were successfully transformed into STM models. The models were then loaded in an editor, which is available in the EMF suite. The model editor has a model validation option, which confirmed that all the models conformed to the STM.

Next, the model steps, the name and number of services, the orchestration's operations, and their execution order were checked and compared to their source model. The results indicated that the transformation algorithms managed to generate STM models according to the original information and respecting the original operation sequence. Both transformation algorithms have the same computational complexity O(n). Both algorithms require computational time equivalent to the number of operations consumed by the SOA orchestration.

By comparing the transformation results of the ImageScraper Orchestration, from both LiteBPEL and BPMN, the transformation results are different. The number of steps and their model complexity is different, which can be verified by the number of lines on the resulting STM models (82 lines from LiteBPEL, 71 lines from BPMN). Both models represent the same operation sequence, which generates similar test cases. However, the difference between the original models was transferred to the generated STM models.

The JSON files were successfully generated from the M2T transformations. The transformations respected the original operation sequence of the original models. The JSON files were used as input for the test case generation on the SAMBA framework; all of them were used successfully in the test case generation. Table III presents the overview of the results obtained in the case study.

TABLE III
OVERVIEW OF CASE STUDY RESULTS

Orchestration	Description	Lines of code		Model Val.	Test Gen.
		Original	STM		
ImageScraper	LiteBPEL	314	82	yes	yes
HyperTimeTable	LiteBPEL	179	65	yes	yes
FeedReader	LiteBPEL	83	33	yes	yes
ImageScraper	BPMN	513	71	yes	yes
Customers	BPMN	516	39	yes	yes
HiringProcess	BPMN	130	30	yes	yes

All the artifacts generated and the transformations used in this case study are available in [22], while the SOA metamodel is available in a separate git repository [23].

C. Research Answers

We now discuss answers to the research questions from Section III.

What is the typical information necessary to generate an input model to a standard MBT tool that targets SOA applications?

Comparing the related works, we identified some of the differences between MBT tools and model specifications. According to Utting et al., the model specification is divided into three dimensions: Scope, Characteristics, and Paradigms [15]. We decided to focus on the model specification because the other dimensions of the MBT approaches do not directly influence the information required from the SUT.

What are the challenges related to the transformation from different SOA description models to a generic model?

The two notations used in the case study, BPMN and BPEL, have entirely different approaches to describe business processes. BPMN describes the sequence of operation by a simple parameter, which indicates the next FlowNode to be computed. The BPEL notation defines clusters with Flow and Sequence Activities, which indicate how the operations should be executed (parallel or sequence).

The challenges of realizing the transformation code for each SOA description are specific to each notation. The transformation complexity varies according to the differences between model paradigms of the source and target format. The BPEL notation was designed to describe SOA orchestrations. However, BPMN can describe both orchestrations and choreographies. In the current implementation of the STM, the application architectural paradigm is abstracted. It is possible to perform a similar experiment for SOA choreographies using the same transformations developed for this case study.

D. Threats to Validity

The first threat is that STM was designed to attend the subjects' requirements in the case study, because of that, we consider only two SOA metamodels (BPEL and BPMN). We decided to keep this small group of notations to reduce the number transformations required by the case study.

Many related works perform the case study with SOA applications described in UML and rely on MBT tools compatible with the UML notations. The case study on this work only used one MBT tool, and its not UML compatible. This is the second threat to the validity of this work since we can not correctly evaluate the challenges related to the transformation from the STM to different input models.

VII. Related Works

There are many works in the literature that present MBT approaches [32]. In order to filter papers, we defined search topics based on the main features of our work. Obligatory inclusion criteria for the selected papers are: i) exploits MDE techniques for MBT; ii) relies on a metamodel; iii) performs automated generation of test cases; iv) exercises the technique on a case study.

Table IV presents the identified works according to the defined selection criteria. Despite the fact that these topics are met in various related works, to the best of our knowledge, no works are exploiting MDE to enable interoperability of model-based testing (of orchestration), such that incompatibilities due to different orchestration languages and test models are overcome, which is the main contribution of this paper.

The work from S. R. Dalal et al. [33] presented an MBT approach that uses a data model to generate test cases. The approach automatically generates tests that comply with the test requirements. The data model is easy to be updated, which grants agility on the test case generation in response to changes. However, the approach requires an oracle and skilled testers.

Krenn et al. [34] proposed a model-based mutation testing (MBMT) approach. The system model is based on UML statecharts, class diagrams, and instance diagrams as input for the test case generation. The mutants of the modeled system are generated, then a black-box test case generation tool, specialized in fault-based test case generation, tries to kill the mutants.

Pérez et al. [35] introduce an MBT tool that generates abstract test cases from conceptual models of a system. It uses the UML class diagram and the state transition diagram to represents the structure and behavior of the SUT. Main results showed that the approach required knowledge in the diagram specification, but it was capable of generating large numbers of functional tests automatically.

The "Model and Inference Driven Automated testing of Services architectures (MIDAS) – Testing as a Service" [36] platform performs MBT on SOA orchestrations. The approach relies on the MIDAS Domain Specific Language (DSL), a selection of concepts from UML and the UML Testing Profile (UTP).

Hernandez et al. [37] present an MBT case generation technique to validate web applications. The paper relies on the use of model-driven software development (an MDE branch) to generate test scripts for testing tools. The work presents a platform-independent metamodel, based on a UML 2.0 profile, designed to represents user interfaces of a web application.

TABLE IV
Selected related works according to the inclusion criteria.

Approach	i	ii	iii	iv	Agile dev.	end-to-end
S. R. Dalal et al. [18]		x	x	x		
Krenn et al. [19]		x	x	x		
Pérez et al. [20]	x	x	x			
MIDAS [21]	x	x	x	x	x	x
Hernandez et al. [22]	x	x	x			
Bentakouk et al. [23]	x	x	x	x		x
Meryem et al. [24]	x	x	x	x	x	x

The metamodel provides abstractions for the visual elements of HTML pages that are relevant to the test script generation.

Bentakouk et al. [38] present an automatic testing approach for BPEL orchestrations. The proposed formal framework relies on the manual translation of the orchestration specifications into a formal model, named Symbolic Transition System (STS). The STS is then used to generate a Symbolic Execution Tree (SET). The approach was automated by prototypes written in Java and Python, serving as proof of concept. A BPEL orchestration (described in XML) was converted to a UML Profile (STS), then the UML2CSP tool converted the STS to communicating sequential process (CSP) model that is used to generate the SET. Authors advocate that transformation rules can be defined from other languages with workflow features (UML, BPMN) and accordingly provide the software architect with a richer specification environment.

Meryem et al. [39] proposed an MBT approach within the Scrum agile process to generate test cases. The approach implements two cartridges for the AndroMDA framework: a model-to-model (M2M) and a model-to-text (M2T) transformation. The M2M transformation generates a test model from a design model. First, a user story is chosen, then it is transformed into a platform-independent design model, described by a sequence diagram, and finally into a platform-independent test model, described in the UML2 Test profile. The M2T transformation generates test case codes for a tool called TestNG, which is a Java unit testing framework.

Overall, UML is the most popular language among all the MBT approaches [40], and it is also the most popular choice among the related works. Nonetheless, some approaches relied on other metamodels [33], [38]. In general, the model can imply restrictions on the test case generation technology, and test execution methods. This is the main limitation that we aim to overcome with our approach, by improving the capacity of the testing framework to operate seamless with different model languages or MBT tools.

VIII. Conclusions

In the context of services orchestration, this paper discusses the application of Model-Driven Engineering (MDE) principles as meta-modeling and model transformation to improve the interoperability of model-based testing (MBT) tool. The research produced the SOA Testing Metamodel (STM), which stores the business process behavior and the information required to generate input models for a MBT framework.

The proposed approach is implemented and evaluated on a case study where multiple orchestrations are expressed in the BPEL and BPMN languages, showing how intermediate artifacts are properly produced and that GraphWalker test cases are appropriately generated and executed.

Given the positive results achieved in this work, the next steps will focus on broadening the scope of the STM from SOA orchestration. First, we plan to adapt STM to include choreographies and real-time systems. The goal is to enrich the pool of transformations available, in order to evaluate different kinds of distributed systems at runtime, like cyber-physical systems.

With respect to services, the most obvious difference is that cyber-physical systems include physical interfaces, and the possible interactions are less structured than in orchestrations, although proposals for solid and relied-upon cyber and physical interfaces exist [41]. We believe that a revised STM and an different MBT framework able to operate with simulated environments as digital twins [42], can bring a relevant contribution to the testing of cyber-physical systems as well.

REFERENCES

[1] Erl, Thomas. Service-oriented architecture. Vol. 8. New York: Prentice hall, 2005.

[2] He, Wu, and Li Da Xu. "Integration of distributed enterprise applications: A survey." IEEE Transactions on industrial informatics 10.1 (2012): 35-42.

[3] Butzin, Björn, Frank Golatowski, and Dirk Timmermann. "Microservices approach for the internet of things." 2016 IEEE ETFA. 2016.

[4] Varga, Pal, et al. "Making system of systems interoperable–The core components of the arrowhead framework." Journal of Network and Computer Applications 81 (2017): 85-95.

[5] Peltz, Chris. "Web services orchestration and choreography." IEEE Computer 36.10 (2003): 46-52.

[6] Bozkurt, Mustafa, Mark Harman, and Youssef Hassoun. "Testing and verification in service-oriented architecture: a survey." Software Testing, Verification and Reliability 23.4 (2013): 261-313.

[7] Marijan, Dusica, and Sagar Sen. "DevOps Enhancement with Continuous Test Optimization." SEKE. 2018.

[8] Demeyer, Serge, et al. "Evaluating the efficiency of continuous testing during test-driven development." 2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST). IEEE, 2018.

[9] Walgude, A., and Natarajan, S. "World quality report 2019-20." Technical report, Capgemini, Sogeti and HPE, 2020.

[10] Belli, Fevzi, et al. "A holistic approach to model-based testing of Web service compositions." Software: Practice and Experience 44.2 (2014): 201-234.

[11] Topçu, Okan, et al. "Distributed Simulation - A Model Driven Engineering Approach". Springer, 2016.

[12] Leal, Lucas, Andrea Ceccarelli, and Eliane Martins. "The SAMBA approach for Self-Adaptive Model-BAsed online testing of services orchestrations." 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC). 2019.

[13] Schmidt, D. C. (2006). "Guest Editor's Introduction: Model-Driven Engineering." IEEE Computer, 39(2), 25.

[14] Da Silva, Alberto Rodrigues. "Model-driven engineering: A survey supported by the unified conceptual model." Computer Languages, Systems & Structures 43 (2015): 139-155.

[15] Utting, Mark, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches." Software testing, verification and reliability 22.5 (2012): 297-312.

[16] Kephart, J. O., & Chess, D. M. (2003). "The vision of autonomic computing." IEEE Computer, (1), 41-50.

[17] Standard, O.A.S.I.S. (2007). "Web services business process execution language version 2.0."

[18] Karl, Kristian. "Graphwalker." www.graphwalker.org [accessed: 2020-09-14] (2003).

[19] Abrahamsson, Pekka, et al. "Agile software development methods: Review and analysis." arXiv preprint arXiv:1709.08439 (2017).

[20] Cumberlidge, Matt. "Business process management with JBoss jBPM." Packt Publishing Ltd, 2007.

[21] von Rosing, Mark, et al. "Business Process Model and Notation-BPMN." (2015): 429-453.

[22] Leal, L. (2020). "SAMBA metamodel for MBT online testing." https://github.com/LucasCLeal/SAMBAMM.git [accessed: 2020-09-02].

[23] Leal, L. (2020). "SAMBA metamodel ATL and Acceleo Transformations." https://github.com/LucasCLeal/SambaMMTransformations.git [accessed: 2020-09-02].

[24] Czarnecki, Krzysztof, and Simon Helsen. "Classification of model transformation approaches." Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture. Vol. 45. No. 3. 2003.

[25] Steinberg, Dave, et al. "EMF: eclipse modeling framework." Pearson Education, 2008.

[26] Barr, Earl T., et al. "The oracle problem in software testing: A survey." IEEE Transactions on Software Engineering 41.5 (2014): 507-525.

[27] Jordan, Diane, et al. "Web services business process execution language version 2.0." OASIS standard 11.120 (2007): 5.

[28] Allilaire, Freddy, et al. "ATL-eclipse support for model transformation." Proceedings of the Eclipse Technology eXchange workshop (eTX) at ECOOP 2006, Nantes, France.

[29] Musset, J., Aurelien Pupier Obeo, and Cedric Brun. "Acceleo." http://wiki.eclipse.org/Acceleo (2012).

[30] Delerce-Mauris, C., Palacin, L., Martarello, S., Mosser, S., & Blay-Fornarino, M. (2009). "Plateforme JSEDUITE: une approche SOA de la diffusion d'informations." University of Nice, I3S CNRS, Sophia Antipolis, France.

[31] Abrahamsson, Pekka, et al. "Agile software development methods: Review and analysis." arXiv preprint arXiv:1709.08439 (2017).

[32] Li, Wenbin, Franck Le Gall, and Naum Spaseski. "A survey on model-based testing tools for test case generation." International Conference on Tools and Methods for Program Analysis. Springer, Cham, 2017.

[33] Dalal, Siddhartha R., et al. "Model-based testing in practice." Proceedings of the 21st international conference on Software engineering. 1999.

[34] Krenn, Willibald, et al. "Momut:: UML model-based mutation testing for UML." 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2015.

[35] Pérez, Constanza, and Beatriz Marín. "Automatic generation of test cases from UML models." CLEI Electron. J. 21.1 (2018).

[36] Herbold, Steffen, et al. "The MIDAS cloud platform for testing SOA applications." 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2015.

[37] Hernandez, Yanelis, et al. "A Meta-model to Support Regression Testing of Web Applications." SEKE. 2008.

[38] Bentakouk, Lina, Pascal Poizat, and Fatiha Zaïdi. "A formal framework for service orchestration testing based on symbolic transition systems." Testing of Software and Communication Systems. Springer, Berlin, Heidelberg, 2009. 16-32.

[39] Elallaoui, Meryem, Khalid Nafil, and Raja Touahni. "Introducing model-driven testing in scrum process using U2TP and AndroMDA." International Review on Computer and Software (IRECOS) 12.1 (2017): 30-39.

[40] Neto, Arilo Dias, et al. "Improving evidence about software technologies: A look at model-based testing." IEEE Software 25.3 (2008): 10-13.

[41] Ceccarelli, A., Bondavalli, A., Froemel, B., Hoeftberger, O., and Kopetz, "Basic concepts on systems of systems." In Cyber-Physical Systems of Systems (pp. 1-39). Springer, 2016.

[42] Gabor, Thomas, et al. "A simulation-based architecture for smart cyber-physical systems." IEEE International Conference on Autonomic Computing (ICAC). IEEE, 2016.