Requirements Engineering (2019) 24:119–132
https://doi.org/10.1007/s00766-018-0304-3

ORIGINAL ARTICLE

Automatic test cases generation from business process models

Arezoo Yazdani Seqerloo[1] · Mohammad Javad Amiri[2] · Saeed Parsa[3] · Mahnaz Koupaee[2]

Received: 14 April 2017 / Accepted: 11 July 2018 / Published online: 17 July 2018

Abstract
Traditional test case generation approaches focus on design and implementation models while a large percentage of software errors are caused by the lack of understanding in the early phases. One of the most important models in the early phases of software development is business process model which closely resembles the real world and captures the requirements precisely. The aim of this paper is to present a model-based approach to automatically generate test cases from business process models. We first model business processes and convert them to state graphs. Then, the graphs are traversed and transformed to the input format of the "Spec explorer" tool that generates the test cases. Furthermore, we conduct a study to evaluate the impact of process characterizations on the performance of the proposed method.

Keywords Business process model · Model-based testing · Test case generation · Spec Explorer

1 Introduction

The software testing area has a wealth of techniques that can be used in software development in order to detect software defects [1]. Software testing methods describe how to test software in details and introduce a process to test validity and verifiability of software. This process starts with test planning and continues with designing test cases, sketching test cases, preparing for execution and evaluating the status till the test closure [2]. Test case generation is one of the most important tasks in software testing. Indeed, increasing the probability of finding errors using a limited number of test cases that are performed in a short time with minimum effort is a desirable property [3]. A goal is to find a minimum set of test cases that have the highest coverage and maximum fault detection percentage to satisfy the test adequacy criteria of the software under test and at the same time being capable of finding errors in early phases of software development. This paper provides a step to realize this goal.

✉ Mohammad Javad Amiri
amiri@cs.ucsb.edu

[1] Department of Computer Engineering, Tehran University, Tehran, Tehran, Iran

[2] Department of Computer Science, University of California Santa Barbara, Santa Barbara, CA, USA

[3] Department of Computer Engineering, Iran University of Science and Technology, Tehran, Tehran, Iran

While the code-based generation of test cases helps to capture errors at the implementation level, analysis and design misunderstanding and errors remain in code [4]. To solve this issue, software developers produce test cases before generating code through model and system specifications. This strategy is known as model-based testing. Generally, model-based testing (MBT) involves extracting test cases from a model to show the expected system behavior in the format of a behavioral model [5]. The main activities in test design are creating the behavioral model of the system and determining how it relates to the original system. Model-based testing consists of four main steps: (1) modeling, (2) test generation, (3) test concretization and (4) test execution [6, 7].

The majority of development methodologies treat requirements activities separately from the development activities [8]. Generating test cases from design models and even use-case diagrams ends in test cases that are unable to capture errors in the requirement extraction phase. The requirements extraction phase is the most critical phase of the software development lifecycle (SDLC) and any wrong or missing requirements lead to wrong or incomplete product; no matter how good the subsequent phases are [9]. On the other hand, creating test cases based on the requirements will maximize the independence between the model and the system under test [10]. Also, requirement specifications are the origin of the information that realizes the functionality of a software

[11]. The goal of this paper is to introduce a method to generate test cases that cover requirement specifications.

Organizational models show the structure and behavior of an enterprise and are very useful in helping developers properly understand the organizational environment and the requirements that the system must fulfill. Therefore, a good knowledge of the application domain is critical to be able to succeed in requirements elicitation [12]. Business process models are widely used in requirement engineering area to extract requirements [13, 14]. Although requirements engineering is the bridge between enterprise and system domains, most of the research in this area is still solution oriented, which does not address the real problems of the organization. As a consequence, since the enterprise is not correctly analyzed, the information system may not meet expectations and business/IT alignment will not be achieved [12]. In this paper, we study the problem of test case generation and describe a business process-driven approach that allows generation of test cases from software requirements to support the operations of an enterprise and assure business/IT alignment.

A business process model is a set of related and collaborative activities or tasks that produce a specific service or product [15]. We model processes to have a comprehensive understanding of the system, make process improvement easy, or execute them [16]. Although some research uses process-oriented approaches to generate test cases [17], in these researches process models are generated in the software design phase and do not capture the flow of activities in organizations and businesses.

The main contribution of this paper is to propose a technique to generate test cases from requirement specifications using business process models. To this end, execution paths are extracted from business processes. These paths resemble the real word and can capture requirements more precisely. Based on this, in the proposed method, the business process model is first generated using the BPMN 2.0 standard [18] and transformed to a state graph. Then, behavioral scenarios are extracted from the state graph, and finally using the Spec Explorer software [19], test cases are generated from the graph. In this model, test sequences are extracted from business process models with the highest coverage. A main feature of the model-based testing is the ability to generate tests from models for different test purposes. This is usually based upon certain metrics that measure the adequacy of the test suites toward addressing a given test purpose [20]. The proposed method saves time and costs of generating test cases while maintaining coverage.

This paper is organized as follows. Section 2 reviews the related work of automatic test case generation. Section 3 describes the steps of the proposed method. In Sect. 4, experimental evaluation is established for the proposed method, and finally, Sect. 5 concludes the paper.

2 Related work

Test case generation techniques can be either model based or manual. We will briefly discuss existing methods in each category.

Model-based methods mainly generate test cases from a sequence, use-case, or activity diagram. Nayak and Samanta [21] proposed an approach of synthesizing test data from the information embedded in class and sequence diagrams using object language constraints (OCL). In their approach, sequence diagrams are annotated with attribute and constraint information derived from a class diagram and object language constraints and then mapped onto a structured composite graph called SCG. The test specification is then generated from a structured composite graph in two steps. First, a finite set of scenarios which are complete paths starting from the initial node to a final node is generated from SCG, and then the test input satisfying all the constraints along the path is found. Since their method generates test cases from design models, any possible errors in earlier phases, e.g., requirement elicitation, cannot be captured. Sarma and Mall [22] transform use-case and sequence diagrams to use-case diagram graph (UDG) and sequence diagram graph (SDG), respectively, and then integrate these two graphs to form a system-testing graph (STG). The system-testing graph is then traversed to generate test scenarios using state-based transition path coverage criteria. This approach finds three important faults, which usually occur during system development: use-case initialization faults, use-case dependency faults and operational faults. The first two categories can be covered using UDG, whereas the last can be covered using SDG. In [4], sequential messages and interactions between objects are also explored using sequence and interaction overview diagrams of UML 2. In [23, 24], several coverage criteria based on use-case diagrams are proposed. These criteria are use-case step coverage, use-case branch coverage, use-case scenario coverage, use-case boundary body coverage and use-case path coverage. To generate test cases from UML activity diagrams in [25], activity diagrams are transformed to a grammar called activity convert (AC) grammar, and then the activity convert grammar is used to generate the test cases.

Traditional software testing methods generate test cases manually through system analysis [26] resulted in incompetence and lack of test cases integrity. These methods can be divided into specifications-based, design-based and code-based methods. Among design-based methods, UML state chart [27, 28] is mainly used to generate test cases; however, faults in requirements analysis and business modeling are not taken into account. In [29], test cases are generated by transforming state design diagram to an expanded finite state machine (EFSM) where the resulting graph has no

hierarchy or synchronization. A combination of state and sequence diagrams is used in [30] to generate test cases. In this method, the main information is extracted from the sequence and state diagrams. The main challenges are the lack of automatic test cases generation and incomplete coverage of all paths in the system under test. [31] also generates test cases using state and sequence diagrams. In [32], authors take advantage of the requirements models to generate test cases using the Communication Analysis (CA). Transformation rules are defined to facilitate the generation of test models, and refinement rules are defined to obtain the abstract test cases from the test model. This method lacks the automatic generation of test cases. In [33], a combination of states model and activities of the target system is used to produce the state-activity diagram (SAD). This diagram is then used to generate test cases that cover states and activities. However, the possible differences between states model and the real requirements are not considered.

Ignoring requirement analysis in design-based methods makes those methods unable to capture any possible errors or misunderstanding that can occur in the analysis phase. In [32], authors try to capture requirements manually resulting in incompetence and lack of integrity of test cases. In [34], an extended control flow graph (XCFG) is proposed to represent a BPEL process. To generate test cases, first, they introduce an algorithm to generate sequential test paths from the XCFG according to branch coverage criterion. Then, the sequential test paths will be combined into concurrent test paths based on various BPEL structures, and finally, a constraint solver is used to generate test data.

Our work is also closely related to business process transformation methods wherein these methods process models are transformed to either an executable code like BPEL [35] or an analysis model like use-case diagram [36–38]. In [39] also a mapping from business process model to states chart is presented where for each element in process model a translation in state chart is provided. Although the mapping covers all elements, it suffers from lack of optimization that makes the resulting state graph very large in terms of the number of states.

3 Test case generation

The proposed method generates test cases from business process models. Since the selection and construction of a behavioral model is a design task, it is assumed that after understanding the target system, the development team models the system using business process models. Basically, to create the models, workflow patterns should be used. Using these patterns affects the coverage and the number of test cases. In the next phase, business processes are modeled and mapped to state graphs. In fact, through this mapping, a simple graph is established and the complexity of business processes is reduced. Since the finite state machines describe the overall behavior of an object, test cases can be generated from them. In this graph, the ultimate goal is to meet each state and test each event at least once. Finally, in the last step, the test cases are generated using Spec Explorer tool. An outline of the proposed method is shown in Fig. 1. As it can be seen, business processes are modeled and mapped to state graphs, then test cases are generated and finally, the behavior of the system is examined according to these test cases and the test results shown.

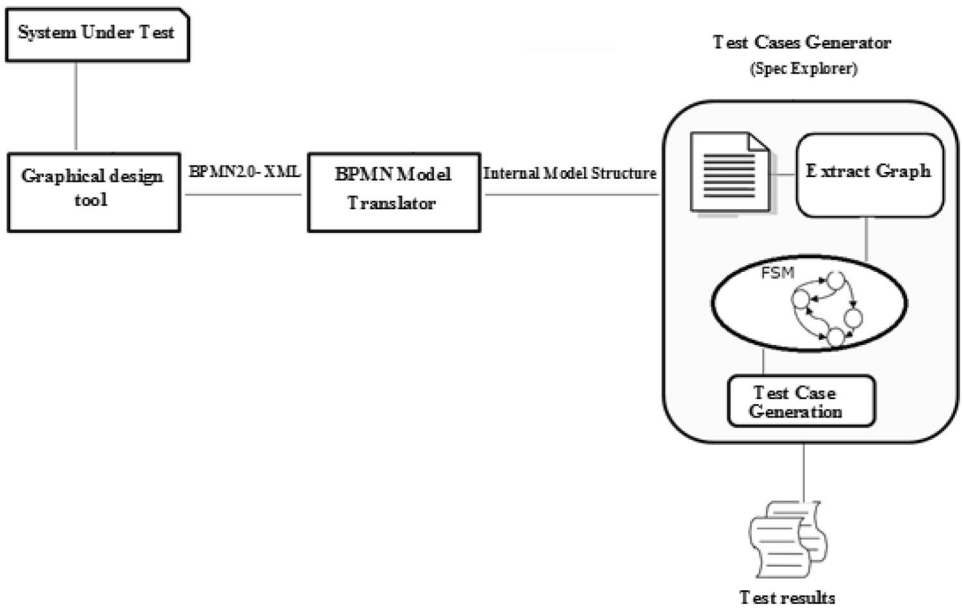

Fig. 1 The proposed model for test case generation

To describe the proposed method, a *Seminar Organizing* process is considered as a running example which is explained as follows.

3.1 Running example

Consider a *Seminar Organizing* process at a university. The process starts with preparing the seminar description. Then, the description will be published and the seminar will be announced. By performing early registration and depending on the number of participants, the organizer decides to confirm or cancel the seminar. In case of confirmation, they prepare for the seminar and perform late registration simultaneously. When these two tasks are performed the seminar can be run.

Figure 2 shows the *Seminar Organizing* process using Business Process Model and Notation (BPMN 2.0) standard. Simplicity, expressiveness and high usage have been the main advantages of this standard, compared with other methods of business process modeling [40, 41]. Nonetheless, every business process modeling language that supports basic workflow notions, e.g., different types of node and edges, can be used in the proposed method.

3.2 Modeling business processes

In the first step, business processes are modeled using BPMN [18]. Each business process is a graph consisting of a set of tasks, gateways, events and connectors [42]. In this paper, we adopt the formal definition of a process from [43]. We focus on one type of edges corresponding to "sequence flow" in BPMN, and three types of nodes: "event," "activity" and "gateway." Furthermore, we consider two classes of *start* and *end* events. An *activity* represents an atomic unit of work. *Gateways* are used to control the divergence and convergence of sequence flows. There are four kinds of frequently used gateways in BPMN: *choice*, *merge*, *split* and *join*. Choice and merge gateways allow a flow in a process to follow one of several alternatives (choice) or choose only one flow from possibly several incoming edges to continue (merge). Split and join gateways, on the other hand, forward a flow to every outgoing edge for parallel execution (split) or synchronize flow from all incoming edges and combine them into one (join). Let's assume the existence of countably infinite, pairwise disjoint sets of U_A and U_G as activities and gateways, respectively.

Definition A business process is a tuple $P=(V, s, f, E, \tau)$ such that

- $V \subseteq U_A \cup U_G \cup \{s, f\}$ is a finite non-empty set of control flow nodes where s and f are the start and final nodes (resp.),
- $E \subseteq (V-\{f\}) \times (V-\{s\})$ is a finite set of control flow edges such that
- s has one outgoing edge and no incoming edges,
- f has one incoming edge and no outgoing edges,
- each node in $(V \cap U_A)$ is an activity node with one incoming and one outgoing edge,
- for each node in $(V \cap U_G)$, the number of incoming edges plus the number of the outgoing edges is at least three,
- τ: $(V \cap U_G) \rightarrow \{G_A, G_X\}$ is a mapping that assigns each gateway a type being AND (Split/Join) Gateway G_A and XOR (Choice/Merge) gateway G_X.

Figure 2 shows the *Seminar Organizing* process, where s and f are the start and final nodes (resp.), t_i's $(1 \leq i \leq$ T9) are the activity nodes, and g_i's $(1 \leq i \leq 4)$ are the gateway nodes where $\tau(g_1)=\tau(g_4)=G_X$ and $\tau(g_2)=\tau(g_3)=G_A$.

We define a path as a set of activities (can be empty) that can exist between two gateways, between the start node and a gateway, between a gateway and an end node, or between the start node and a final node.

Definition Given a process $P=(V, s, f, E, \tau)$, and a sequence of nodes $v_0, v_1, v_2, \ldots, v_n, v_{n+1}$ where for each $i \in [0 \ldots n+1]$,

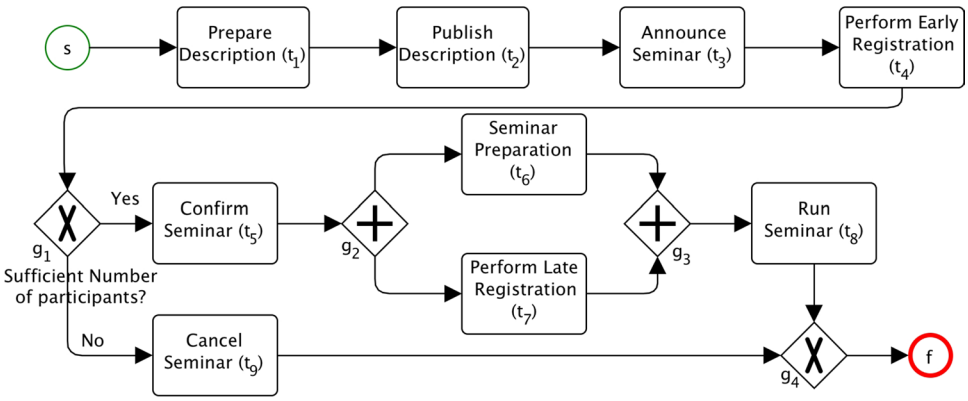

Fig. 2 BPMN process model for Seminar Organizing

$v_i \in V$. A *path* φ is a sequence $v_1, v_2, \ldots, v_n$ such that for each $i \in [1 \ldots n]$, v_i is a node in $(V \cap U_A)$, v_0 is a node in $(V \cap U_G)$ U $\{s\}$ and v_{n+1} is a node in $(V \cap U_G)$ U $\{f\}$. Let φ^0 and φ^z denote the first node (activity) and the last node (activity) of the path φ, respectively.

Depending on v_0 and v_{n+1}, 4 types of paths can be defined in a process:

1. v_0 is the start node and v_{n+1} is the final node,
2. v_0 is the start node and v_{n+1} is a node in $(V \cap U_G)$
3. v_0 is a node in $(V \cap U_G)$ and v_{n+1} is the final node
4. Both v_0 and v_{n+1} are nodes in $(V \cap U_G)$.

Given a process $P = (V, s, f, E, \tau)$, node v_i *derives* node v_j, denoted as $\pi(v_j) = v_i$, iff $(v_i, v_j) \in E$. In other words, an outgoing edge of v_i can be an incoming edge of v_j.

The following paths are defined for the running example (Fig. 2).

- Prepare Description, Publish Description, Announce Seminar, Perform Early Registration $[t_1, t_2, t_3, t_4]$ (type2).
- Confirm Seminar $[t_5]$ (type4)
- Cancel Seminar $[t_9]$ (type4)
- Seminar Preparation $[t_6]$ (type4)
- Perform Late Registration $[t_7]$ (type4)
- Run Seminar $[t_8]$ (type4).

Business process models not only have to capture business requirements precisely but also are required to ensure successful workflow execution. Therefore, we need to verify the correctness of business processes before the process models are implemented [44]. One accepted notion of correctness is structuredness. A structured process consists of m sequential blocks, $B_1, \ldots, B_m$. Each block B_i is either an activity or a composite block. A composite block consists of n parallel, conditional, or loop sub-fragments where each of them is again either an activity or a composite block.

In order to construct a well-structured process model, we define a set of incremental process update operations. These update operations extend a process by replacing an activity with a basic process block, i.e., a sequence of two activities, choice block, parallel block, or loop block.

Definition Let $P = (V, s, f, E, \tau)$ be a business process. δ is an *incremental update operation* on P that transforms P to $P' = (V', s, f, E', \tau')$. Assuming (u, a) and (a, v) are the incoming and the outgoing edges of the activity a, four different types of update operations are defined as follows:

a) AddActivity(activity a, activity a')

Replaces activity a with a sequence of activities a and a' where a' is a new activity in U_A–V.

- $V' = V \cup \{a'\}$,
- $E' = E - \{(a, v)\} \cup \{(a, a'), (a', v)\}$ *and*
- $\tau' = \tau$.
- *b) AddChoiceBlock(activity a, activity a', (choice) gateway g_1, (merge) gateway g_2)*

Replaces an existing activity a with a choice block containing a new choice gateway g_1, a new merge gateway g_2, and two mutually exclusive activities a and a' where a' is a new activity in U_A–V.

- $V' = V \cup \{a', g_1, g_2\}$,
- $E' = E - \{(u, a), (a, v)\} \cup \{(u, g_1), (g_1, a), (g_1, a'), (a, g_2), (a', g_2), (g_2, v)\}$,
- $\tau'(g_1) = \tau'(g_2) = G_X$, and for each gateway node g in $(V \cap U_G)$, $\tau'(g) = \tau(g)$.
- *c) AddParallelBlock(activity a, activity a', (split) gateway g_1, (join) gateway g_2)*

Replaces an existing activity a with a parallel block containing a new split gateway g_1, a new join gateway g_2, and two parallel activities a and a' where a' is a new activity in U_A–V.

- $V' = V \cup \{a', g_1, g_2\}$,
- $E' = E - \{(u, a), (a, v)\} \cup \{(u, g_1), (g_1, a), (g_1, a'), (a, g_2), (a', g_2), (g_2, v)\}$,
- $\tau'(g_1) = \tau'(g_2) = G_A$, and for each gateway node g in $(V \cap U_G)$, $\tau'(g) = \tau(g)$.
- *d) AddLoopBlock(activity a, activity a', (merge) gateway g_1, (choice) gateway g_2)*

Replaces an existing activity a with a loop block containing a new merge gateway g_1, a new choice gateway g_2, and two activities a and a' where a' is a new activity in U_A–V.

- $V' = V \cup \{a', g_1, g_2\}$,
- $E' = E - \{(u, a), (a, v)\} \cup \{(u, g_1), (g_1, a), (a, g_2), (g_2, a'), (a', g_1), (g_2, v)\}$,
- $\tau'(g_1) = \tau'(g_2) = G_X$, and for each gateway node g in $(V \cap U_G)$, $\tau'(g) = \tau(g)$.

Definition Given a business process $P = (V, s, f, E, \tau)$, P is a well-structured business process if there is a sequence of incremental update operations $\delta_1, \delta_2, \ldots, \delta_n$ where $\delta_n \delta_{n-1}, \ldots, \delta_1(P') = P$ *and* $P' = (\{s, a, f\}, s, f, \{(s, a), (a, f)\}, \tau)$ is an atomic schema.

The running example is already well-structured.

3.3 Creating state graph from business process model

In this subsection, an algorithm to create the state graph from process models is introduced. We first define state graph which is needed later to generate test cases.

Definition A *State graph* is a tuple $G=(N, T, s^{\epsilon}, s_{\gamma})$ such that

- N is a non-empty set of nodes indicating states,
- $T \subseteq N \times N$ is a finite set of state transitions,
- $s^{\epsilon} \in N$ is the initial state, and
- $s_{\gamma} \in N$ is the final state

To create a state graph, we traverse the process model and consider gateways and paths as states and state transitions, respectively. To optimize the resulted state graph, if a parallel block has no nested block, it (both split and join gateways and both paths from the split to the join gateway) is considered as one state of the graph. Also, since the split and choice gateways have more than one incoming edge, a counter is assigned to each gateway to count the number of visited incoming edges. The algorithm begins with visiting the start node and adding a state s^{ϵ} to the corresponding state graph. Then, the process is traversed to visit other nodes. Three situations can happen:

1. The visited node is an activity node: the status of the node is changed to *visited* and traversing continues.
2. The visited node is an end event: the corresponding state s_{γ} is added to the graph and *FindEdge* function is called to add the related edge. *The FindEdge* function finds the related edge based on the type of an input node. It has two inputs: a state s and a node v. If the node v is connected to the start node, then we connect s^{ϵ} (initial state) to s in the state graph; otherwise, we find the state corresponding to the predecessor node of v and connect that state to s.
3. The visited node is a gateway:

(a) If the type of gateway is a choice (it has only one incoming edge), we change the status of the gateway to *visited*, add a state to the state graph, and call *FindEdge* function.
(b) If node is a split gateway and there is no other gateway through the path between the split and the corresponding join gateway, then both gateways and all nodes on both existing paths between them are marked as *visited* and a state is added to the graph; otherwise, the split gateway is marked as *visited* and a state is added to the graph. In both cases, the *FindEdge* function has to be called.
(c) For join and merge gateways, the counter is incremented and if the counter is equal to two (the number of incoming edges of each gateway), the gateway is marked as *visited*. Then, the corresponding state is added to the graph and by calling *FindEdge* function the edge is added.

For example, let's consider a loop block with merge gateway g_1 and choice gateway g_2 in a process. When the process is traversed from the start node, we first visit the merge gateway (in contrast to choice blocks where we first visit choice gateways). When the merge gateway g_1 is seen, a corresponding state and an edge from the related state are added to the graph. Then, the algorithm proceeds to the outgoing path to find the next gateway. There may be some gateways on the path from g_1 to g_2 in the process and for all those gateways, corresponding states will be added to the graph, but finally, when the corresponding choice gateway g_2 is seen

due to the well-structuredness of the process, there must be a path from g_2 to g_1.

Algorithm 1 shows

cess models.

Algorithm1: Mapping Business Process Model to Graph
Input: A well-structured Process P = (V, s, f, E, τ)
Output: A state graph G = (N, T, s^e, s_y)

1. Let *G* be the empty graph (with no nodes, no edges)
2. *m = 0, s = 1*
3. Let $count_g = 0$, **for each** gateway node $g \in (V \cap U_G)$
4. Let *s* be a start event in *P*
5. Tag *s* as visited
6. Add *s* as s^e to *G* as the initial state
7. Let *v* be the node in *V* such that $\pi(v) = s$ *//v can be derived from s*
8. Tag *v* as new
9. **while** there is a new node *v* in *P* **do**
10. *//case1: v is a task node*
11. **if** *v* is a node in $(V \cap U_A)$ **then**
12. Change *v*'s tag to visited
13. Let *v'* be the node in *V* such that $\pi(v') = v$
14. **if** visited(*v'*) = false **then**
15. Tag *v'* as new
16. **endif**
17. *//case2: v is an end event*
18. **elseif** *v* is an end event in *F* **then**
19. Change *v*'s tag to visited
20. Add s_y to state graph *G*
21. **call** *findEdge(s_y,v)*
22. *//case3: v is a gateway*
23. **elseif** *v* is a gateway node in $(V \cap U_G)$ **then**
24. **if** *v* is a *split* or a *choice gateway* **then**
25. Change *v*'s tag to visited
26. Add s^m to state graph *G*
27. **call** *findEdge(s^m,v)*
28. *m++*
29. **if** $\tau(v) = G_A$ and *these is a* gateway node *u in* $(V \cap U_G)$ s.t $\tau(v) = \tau(u)$ and for each outgoing path $\varphi_o = v_i v_{i+1} \ldots v_k$ from *v*, $\pi(u) = v_k$ or $\pi(u) = v$ (in case that path is empty) **then** *//parallel paths have only activities*
30. **for each** node v_i $(j \le i \le k)$ in each φ_o from *v* **do**
31. Change v_i's tag to visited
32. **endFor**
33. Change *u*'s tag to visited
34. Let *v'* be the node in *V* such that $\pi(v') = u$
35. **if** *visited(v') = false* **then**
36. Tag *v'* as new
37. **endif**
38. **else**
39. **for each** node *v'* s.t $\pi(v') = v$ **do**
40. **if** *visited(v') = false* **then**
41. Tag *v'* as new
42. **endif**
43. **endfor**
44. **endif**
45. **else** *//join or merge gateway*
46. $count_v$ ++
47. **if** $count_v = 1$ **then** *//visit the gateway for the first time*
48. Add s^m to state graph *G*
49. **call** *findEdge(s^m,v)*
50. *m++*
51. **else**
52. Let s^m be the state corresponding to *v*
53. **call** *findEdge(s^m,v)*
54. **if** $in_v = count_v$ **then**
55. Change *v*'s tag to visited
56. **endif**
57. **endif**
58. Let *v'* be the node in *V* such that $\pi(v') = v$
59. **if** *visited(v') = false* **then**
60. Tag *v'* as new
61. **endif**
62. **endif**
63. **endwhile**

64. ***findEdge(s,v)***
65. **Begin**
66. Let *p* be the node in *V* such that $\pi(v) = p$ *//p is the last visited node*
67. **if** *p* is the *start* node *s* **then**
68. Add *edge(s^e, s)*
69. **elseif** *p* is a *gateway* node in $(V \cap U_G)$ **then**
70. Add *edge(s^p, s)* where s^p is the state corresponding to *p*
71. **else** *// p* is an *activity* node in $(V \cap U_A)$ **then**
72. Let φ_a be the path s.t φ_a^z is *p* *//p is the last node of* φ_a
73. Let *k* be the state in *N* such that $\pi(\varphi_a^0) = k$ *//k is the start event or a gateway*
74. Add *edge(s^k, s)* where s^k is the state corresponding to *k*
75. **endif**
76. **End**

Table 1 Mapping information table for object states

Transition/Action/Guard	Corresponding edge in the graph
Entry data	A
Few participants/CancelSeminar	B
Enough participants/ConfirmSeminar	C
RunSeminar	D
End	E

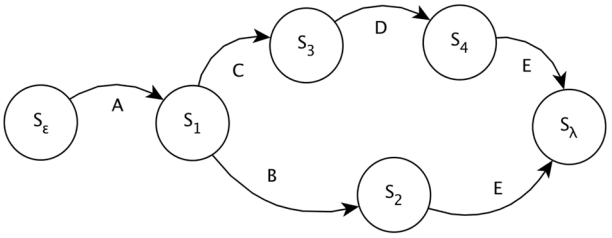

Fig. 3 State graph of the Seminar Organizing process

Figure 3 indicates the state graph generated from the *Seminar Organizing* process model using the Algorithm 1.

The mapping between the transaction, action and condition with the edges of the graph is shown in Table 1.

3.4 Mapping state graph to the input model of Spec Explorer

This subsection provides a brief introduction to the Spec Explorer tool which is used to generate test cases followed by a description of the mapping from state graph to the input model of the tool. Spec Explorer is a tool for testing object-oriented software systems. In Spec Explorer, the behavior of the system is described by a model program written in the language Spec#, an extension of C#. A model program defines the state variables and the update rules of an abstract state machine. The tool explores the machine's states and transitions with techniques similar to those of explicit state model checkers. This process results in a finite graph that is a representative subset of model states and transitions. Spec Explorer uses a state exploration algorithm [45] that is briefly explained as follows:

1. In a given model state (starting with the initial state) determine those invocations (action/parameter combinations) which are enabled by their preconditions in that state,
2. Compute successor states for each invocation,
3. Repeat until there are no more states and invocations to explore.

To map the state graph to the input model of Spec Explorer, the input data, which is needed to choose different states should be passed to the tool. The mapping is performed automatically; each state/transition in the state graph is mapped to an action/transition in the Spec input model and if there is a condition to enable a transition, the condition is attached to model as a data input parameter. Algorithm 2 shows the mapping from a state graph to the input model of Spec Explorer.

```
[Action] void EnoughParticipants (participants) requires participants >= SufficientParticipants && InsertInfo = True
{ConfirmSeminar = True};
[Action] void FewParticipants (participants) requires participants <SufficientParticipants && InsertInfo = True
{CancelSeminar = True};
```

Fig. 4 Part of the input program for the *Seminar Organizing* process

```
var frontier = {(s, a, t) | s ∈ S^init, (s, a, t) ∈ δ}
var included = S^init
var δ' = ∅
while frontier ≠ ∅ ∧ InBounds
  choose (s, a, t)∈ frontier
  frontier := frontier \ {(s, a, t)}
  if t ∈ included ∨ IncludeTarget(s, a, t)
    δ' := δ' ∪ {(s, a, t)}
    if t ∉ included
      frontier := frontier ∪ {(t, a', t') | (t, a', t') ∈ δ}
      included := included ∪ {t}
```

Fig. 5 Directed search in Spec Explorer

Algorithm2: Mapping State Graph to the input model of Spec Explorer
Input: A state graph G = (N, T, s^e, s_y)
Output: A Spec Explorer Model

1. Let *V* be an empty stack
2. *V.push(s^e)*
3. **while** *V* is not empty **do**
4. *s = V.pop()*
5. **if** *s* has one outgoing edge **then**
6. Let *(s,s')* be the outgoing edge
7. Add *"[Action] void s-s' s= True {s' = True};"* to the model *//s-s' is a name for the action*
8. **endIf**
9. **if** *s* has more than one outgoing edge **then** *//s represents a choice gateway*
10. **for each** outgoing edge *(s,s')* where *cond* is its condition on a variable *var* **do** *//Example: cond: var > 3*
11. Add *"[Action] void s-s'(var) requires cond && s = True {s' = True};"* to the model
12. **endFor**
13. **endIf**
14. **if** *s* has no outgoing edge **then** *//s is the final state (s_y) and the graph is completely mapped*
15. **break**
16. **endIf**
17. **for each** outgoing edge *(s,s') of s* **do**
18. *V.push(s')*
19. **endFor**
20. **endWhile**
21. **for each** variable *var* that is used in any condition statement *cond* **do**
22. Define *var* as an input parameter of the model
23. **endFor**

When the model is created, we traverse all the possible paths to produce test scenarios. We use the DFS algorithm to traverse the model. The goal is to meet all states at least onc e. In the *Seminar Organizing* process, two correct execution paths exist; one is when after the early steps, the number of registered participants is not sufficient and the seminar is canceled, and the other one is when enough number of participants are registered and the seminar is confirmed. Here the main input data is the number of participants which is passed as a parameter. Figure 4 shows the part of the input program which is related to the *EnoughParticipants* and the *FewParticipants* actions. Here a Boolean variable is used for each state. As it can be seen, the *EnoughParticipants* action is performed if the number of participants, which is the input parameter of the action, is equal to or greater than the sufficient number of participants and the *InsertInfo* state is visited. This action changes the *ConfirmSeminar* to true. A similar situation exists for the *FewParticipants* action.

Figure 5 shows the general exploration algorithm of Spec Explorer [45]. It assumes two auxiliary predicates:

- *InBounds* is true if user-given bounds on the number of transitions, the number of states, etc., are satisfied.
- *IncludeTarget*(*s, a, t*) is true for those transitions (*s, a, t*) that lead to the desired target state.

By default, *IncludeTarget* returns true. In the algorithm, the variable *frontier* represents the transitions to be explored

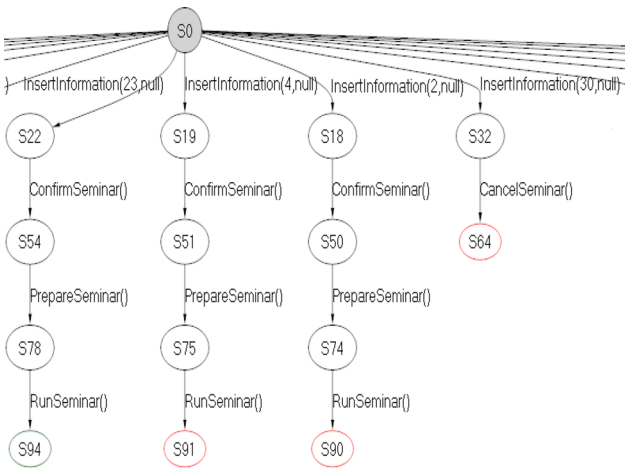

Fig. 6 Part of the input graph to the Spec Explorer Tool

and is initially set to all those transitions which start in an initial state. The variable *included* represents those states of M' whose outgoing transitions have been already added to the *frontier*, and is initially set to the initial states of *M*. The variable *δ'* represents the computed transition relation of the sub-automaton *M'*. The algorithm continues exploring as long as the *frontier* is not empty and the bounds are satisfied. In each iteration step, it selects a transition from the *frontier*, and updates *δ'*, *included* and *frontier*. Upon completion of the algorithm, the transitions of *M'* are the final value of *δ'*.

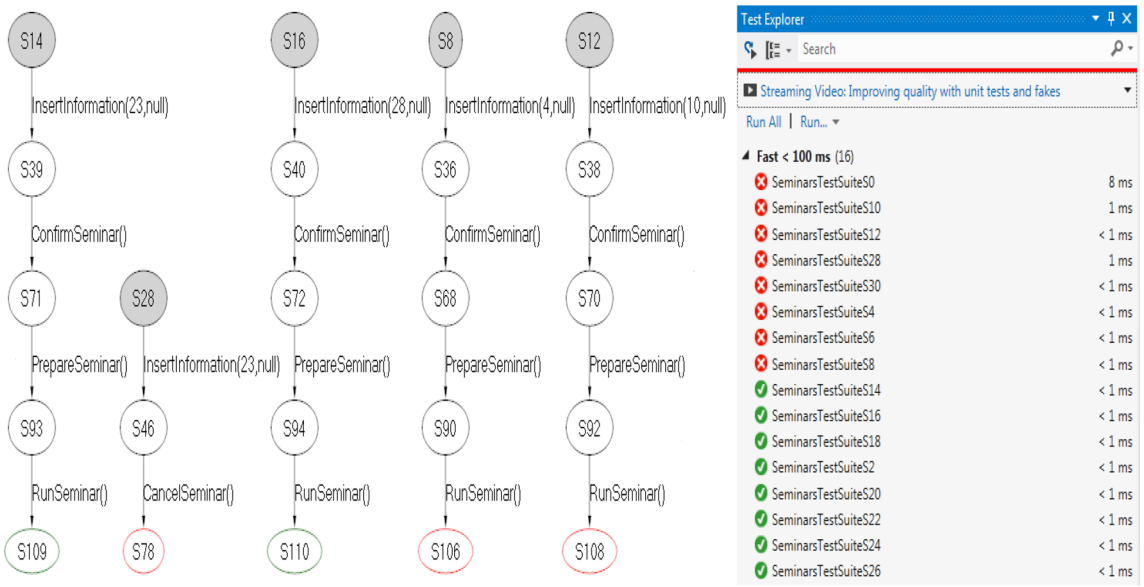

Fig. 7 Graph model and generated test cases for different data

Table 2 University education system processes

Process	#Tasks	#Gateway
Student registration	14	8
Student enrollment in course	6	2
Course plan approval	16	6
Student graduation	10	4
Grades announcement	7	2
Student verification request	11	4
Course auditing	8	2
Transcript verification	12	6

The initial states of *M'* are the initial states of *M*. The states of *M'* consist of all states that are reachable from an initial state of *M'* using the transitions of *M'* [45].

Figure 6 represents a part of input model generated from the algorithm.

3.5 Generating test cases

After creating the graph, the next step is generating test cases. The Spec Explorer tool generates different test cases using the conditions in the model and the domains of the used data. As specified in the definition of well-structured processes, each gateway has at most two outgoing edges (choice/split), therefore, for each condition in the model two possible paths can be created. The number of all possible combinations of gateway conditions is the size of the generated paths which is exponential in terms of choice/split gateways.

Depending on the domain of attributes which are used in the gateway condition, the tool may generate different test cases to capture all behaviors of the model.

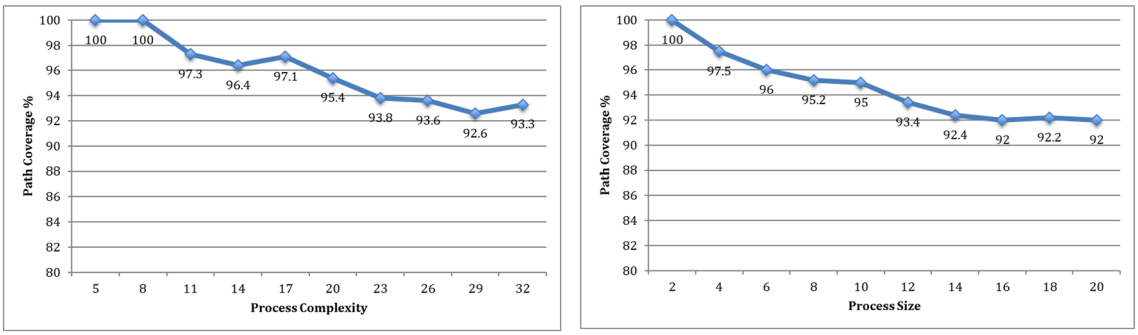

Fig. 8 Path coverage based on process complexity (left side) and process size (right side)

In the running example, we have only one condition which is related to the number of participants. At first glance, four groups of test cases can be generated: (1) having few participants and cancel the seminar, (2) having few participants and confirm the seminar, (3) having enough participants and cancel the seminar, and (4) having enough participants and confirm the seminar where the first and the third groups are desired behaviors of the system. Since the only input data is the number of participants, the tool generated different test cases using different possible numbers. Figure 7 represents a part of the generated test cases in different circumstances.

4 Results and discussion

In this section, three different evaluations are conducted. First, we compare the results of our method and manual test case generation on a software system consisting of eight business processes. Next, to evaluate the impact of process size and process complexity on the path coverage, some arbitrary business processes with different sizes and complexities, are taken into account. Finally, a qualitative comparison between the proposed method and some related work is performed.

We first consider a University education system as a case study. This system was developed several years ago, but we accessed the process models, the implementation and testing steps, and test cases created by domain experts (users/developers) at that time. The goal of this evaluation is to compare the precision and the recall of the test cases that are generated by the proposed method versus the test cases that are generated by users.

The system has the following processes: student registration, course enrollment, course plan approval, student graduation, grades announcement, student verification request, course auditing and transcript verification. Table 2 shows the number of tasks and gateways of these processes.

We generated state graphs and test cases for these processes. Then, based on the number of valid and invalid detected paths, the value of true positive (TP), false positive (FP) and false negative (FP) for the proposed method and system developers are measured. Finally, the precision and the recall of both methods are calculated.

The results are shown in Table 3. The number of existing paths shows the total possible paths in a process model. TP shows the number of valid paths that developers or the proposed method detect as valid paths correctly. FP shows the number of valid paths that are incorrectly detected as invalid paths, and FN shows the number of invalid paths that are incorrectly detected as valid paths. As it can be seen, the average of precision for the users (developers) and the proposed method are 75.75 and 94.37, respectively, which

Table 3 Test results on university education system processes

Process	#States	#Existing Path	Developers TP	Developers FP	Developers FN	The method TP	The method FP	The method FN	Developers precision	Developers recall	The method precision	The method recall
Student registration	10	14	10	3	4	13	0	1	0.77	0.71	1	0.93
Course enrollment	4	4	4	1	0	4	0	0	0.8	1	1	1
Course plan approval	6	7	5	1	2	6	0	1	0.83	0.71	1	0.86
Student graduation	6	6	4	2	2	6	1	0	0.66	0.66	0.86	1
Grades announcement	3	2	2	0	0	2	0	0	1	1	1	1
Student verification request	5	4	3	2	1	4	1	0	0.6	0.75	0.8	1
Course auditing	4	4	2	3	2	4	0	0	0.4	0.5	1	1
Transcript verification	8	9	8	0	1	8	1	1	1	0.89	0.89	0.89

Table 4 Evaluation of the proposed method

Features/ Criteria	Yuan et al. [34]	Abdurazik and Offutt [4]	Granda[32]	Swain et al. [26]	Kansomkeat et al. [31]	Swain et al. [33]	Sokenou [30]	Kim et al. [29]	Bertolino [27] and Offutt and Abdurazik [28]	Abilov and Gomez[39]	Proposed method
Test coverage	M	M	M	L	L	M	M	P	M	P	P
Automatic test cases generation	✓	✓	×	✓	✓	×	×	×	✓	×	✓
Independence and generality of the method	×	✓	✓	✓	✓	✓	✓	×	×	×	✓
Test improvement	✓	✓	×	×	✓	✓	✓	×	×	×	✓
Object-oriented support	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓
Tool support	×	✓	×	✓	×	✓	×	×	✓	×	✓
Business process support	✓	×	×	×	×	×	×	×	×	✓	✓
Automatic determination of test data	✓	✓	×	×	×	×	×	×	✓	×	×

The mark ✓ in the table means the support of the methodology of the desired index; mark × indicates non-support of the related index. "P" indicates "Perfect," "M" indicates "Medium" and "L" indicates "Low"

means the proposed method increases the precision about 24.58%. In addition, the average of recall for the user and the proposed method are 77.75 and 96, respectively, so the proposed method also increases the recall about 23.47%.

In the second set of experiments, we study the impact of process size and complexity on the path coverage. To measure the impact of process size (number of tasks), we choose 10 processes and generate test cases for each of them. The path coverage is then computed. The results show that the accuracy of the method in path coverage is independent of process size. We then, similar to the previous experiment, measure the impact of process complexity (number of gateways) on the path coverage. The results of these two evaluations are shown in Fig. 8. Note that most of the chosen models and their implementations are from existing sources available on the Internet [46, 47].

Finally, we performed a qualitative evaluation. We select a number of most-cited and relevant papers to compare with the proposed method. Most of these papers either use UML diagrams, i.e., use-case, activity, or sequence diagram, to show the behavior of a process [26, 27, 32] or use a process modeling language like BPEL [34]. We also choose 8 metrics for comparison: coverage, automatic test case generation, independency and generality of the method, test optimization and improvement, object-oriented support, tool support, business process support and automatic determination of test data. Table 4 shows the results.

Four kinds of coverage are defined: statement coverage, branch coverage, path coverage and requirements coverage wherein this paper we focus on path coverage. Path coverage is the ratio of the extracted path from the test cases to the existing path in the code. Automatic test cases generation helps to decrease the development cost and time. The third factor is independence and generality of the method which basically shows whether the method can be used for different modeling languages or not. Test improvement refers to the optimizing number of test cases. The three next factors show that a method supports object-oriented development, has a tool, and supports any of the business process modeling languages or not. Finally, automatic determination of test data considers the correlation of conditions and paths.

5 Conclusions and future work

Model-based testing is a useful technique to increase software reliability and reduce costs by generating test cases based on behavioral models of the system. In this study, a method to automatically generate test cases using business process models is presented. Using process model helps to identify test cases of business areas; thus, the test cases will be more compatible with the business requirements and will capture any misunderstanding in future phases in software

development. The algorithm for the conversion of process models to state graphs prepares the model for data production and by navigating the models the preconditions will be determined. These conditions are converted to codes. Then, the tool automatically generates the test cases. Using business workflow, optimization of test cases, preparing an automated method for mapping process modeling into state diagram, and tools support are the main characteristics of this method.

Although this method could detect errors in the domain of business requirements, it does not capture errors in the design and the implementation phases, so using this method alone is not enough to generate test cases. One of the limitations of the proposed algorithm is that it works only for well-structured processes and if the input process is not well-structured, it can't define states correctly and it so captures invalid paths. To solve this issue, we can convert unstructured process to structured ones using algorithms in [48] or [49]. In addition, the algorithm assumes that the conditions attached to the outgoing paths of choice gateways are valid and consistent. Finally, including data could improve the efficiency of test cases generation.

References

1. De Cleva Farto G, Endo AT (2015) Evaluating the model-based testing approach in the context of mobile applications. Electron Notes Theor Comput Sci 314:3–21
2. Backlund A (2010) Utilizing statistics in a model-based testing process. Dissertation, Department of Information Technologies, Abo Akademi University
3. Boghdady PN, Badr N, Hashem M, Tolba MF (2011) Test case generation and test data extraction techniques. Int J Electr Comput Sci (IJECS-IJENS) 11(03):87–94
4. Abdurazik A, Offutt J (2000) Using UML collaboration diagrams for static checking and test generation. In: International conference on the unified modeling language, pp 383–395
5. Nam DH, Mousset EC, Levy DC (2006) Automating the testing of object behavior: a state chart-driven approach. In: Proceedings of world academy of science, engineering and technology, Helsinki, Finland
6. Pretschner A, Philipps J (2005) 10 methodological issues in model-based testing. In: Model-based testing of reactive systems, pp 281–291
7. El-Far IK, Whittaker JA (2001) Model-based software testing. In: Marciniak JJ (ed) Encyclopedia of software engineering. Wiley
8. Badreddin O, Sturm A, Lethbridge TC (2014) Requirement traceability: a model-based approach. In: IEEE 4th international model-driven requirements engineering workshop (MoDRE), pp 87–91
9. Alshazly AA, Elfatatry AM, Abougabal MS (2014) Detecting defects in software requirements specification. Alex Eng J 53(3):513–527
10. Utting M, Pretschner A, Legeard B (2012) A taxonomy of model-based testing approaches. Softw Test Verif Reliab 22(5):297–312
11. Parsa S, Amiri MJ, Ebrahimifard A, Arani MK (2016) Towards a goal-driven method for web service choreography validation. In: 2nd IEEE international conference on web research, pp 66–71
12. De la Vera Gonzalez JL, & Diaz JS (2007) Business process-driven requirements engineering: a goal-based approach. In: Proceedings of the 8th workshop on business process modeling
13. Yu ES (1997) Towards modelling and reasoning support for early-phase requirements engineering. In: Proceedings of the 3rd IEEE international symposium on requirements engineering, pp 226–235
14. Ebrahimifard A, Amiri MJ, Arani MK, Parsa S (2016) Mapping BPMN 2.0 choreography to WS-CDL: a systematic method. J E Technol 7(1):1–23
15. Muehlen MZ, Indulska M, Kamp G (2007) Business process and business rule modeling: a representational analysis. In: Eleventh international IEEE EDOC workshop, pp 189–196
16. Roser S, Bauer B (2005) A categorization of collaborative business process modeling techniques. In: 7th IEEE international conference on E-Commerce technology, pp 43–51
17. Anand S, Burke EK, Chen TY, Clark J, Cohen MB, Grieskamp W, Harman M, Harrold MJ, McMinn P (2013) An orchestrated survey of methodologies for automated software test case generation. J Syst Softw 86(8):1978–2001
18. Business Process Model and Notation (BPMN), Version 2.0. OMG Specification, Object Management Group, August 2013
19. Spec Explorer tool. http://research.microsoft.com/specexplorer, public release January 2005. Accessed 19 May 2018
20. Mohalik S, Gadkari AA, Yeolekar A, Shashidhar KC, Ramesh S (2014) Automatic test case generation from Simulink/Stateflow models using model checking. Softw Test Verif Reliab 24(2):155–180
21. Nayak A, Samanta D (2010) Automatic test data synthesis using UML sequence diagrams. J Object Technol 9(2):75–104
22. Sarma M, Mall R (2007) Automatic test case generation from UML models. In: 10th IEEE international conference on information technology (ICIT), pp 196–201
23. Winter M (1999) Quality assurance for object-oriented software, requirements engineering, and testing wrt requirements specification. Dissertation, University of Hagen
24. Myers GJ, Corey S, Tom B (2011) The art of software testing. Wiley, London
25. Pechtanun K, Kansomkeat S (2012) Generation test case from UML activity diagram based on AC grammar. In: IEEE international conference on computer & information science (ICCIS), vol 2, pp 895–899
26. Swain R, Panthi V, Behera PK, Mohapatra DP (2012) Automatic test case generation from UML state chart diagram. Int J Comput Appl 42(7):26–36
27. Bertolino A (2003) Software Testing Research and Practice. In: Börger E, Gargantini A, Riccobene E (eds) Abstract State Machines 2003. ASM 2003. Lecture Notes in Computer Science, vol 2589. Springer, Berlin, Heidelberg
28. Offutt J, Abdurazik A (1999) Generating Tests from UML Specifications. In: France R, Rumpe B (eds) «UML»'99 — The Unified Modeling Language. UML 1999. Lecture Notes in Computer Science, vol 1723. Springer, Berlin, Heidelberg
29. Kim YG, Hong HS, Bae DH, Cha SD (1999) Test cases generation from UML state diagram. IEEE Proc Softw 146:187–192
30. Sokenou D (2006) Generating test sequences from UML sequence diagrams and state diagrams. GI Jahrestagung 2:236–240
31. Kansomkeat S, Offutt J, Abdurazik A, Baldini A (2008) A comparative evaluation of tests generated from different UML diagrams. In: Ninth ACIS international conference on software engineering, artificial intelligence, networking, and parallel/distributed computing, SNPD'08, pp 867–872
32. Granda MF (2014) An experiment design for validating a test case generation strategy from requirements models. In: 2014 IEEE 4th international workshop on empirical requirements engineering (EmpiRE), pp 44–47
33. Swain SK, Mohapatra DP, Mall R (2010) Test case generation based on state and activity models. J Object Technol 9(5):1–27

34. Yuan Y, Li Z, Sun W (2006) A graph-search based approach to BPEL4WS test generation. In: International conference on software engineering advances, pp 14–22
35. Hauser R, Koehler J (2004) Compiling process graphs into executable code. In: International conference on generative programming and component engineering (GPCE), pp 317–336
36. Cruz EF, Machado RJ, Santos MY (2014) From business process models to use case models: a systematic approach. In: Enterprise engineering working conference, pp 167–181
37. Dijkman RM, Joosten SM, Utopics OF (2002) An algorithm to derive use case diagrams from business process models. In: Proceedings of the 6th international conference on software engineering and applications (SEA), pp 679–684
38. Liew P, Kontogiannis K, Tong T (2005) A framework for business model driven development. In: The 12th IEEE international workshop on software technology and engineering practice (STEP), pp 1–8
39. Abilov M, Gomez JM (2014) Derivation of event-based state machines from business processes. In: Proceedings of the international conference on new trends in information and communication technologies
40. Chinosi M, Trombetta A (2012) A. BPMN: an introduction to the standard. Comput Stand Interfaces 34(1):124–134
41. Amiri MJ, Parsa S, Mohammadzade Lajevardi A (2016) Multifaceted service identification: process, requirement and data. Comput Sci Inf Syst 13(2):335–358
42. Amiri MJ, Koupaee M (2017) Data-driven business process similarity. IET Softw 11(6):309–318
43. Sun Y, Su J (2011) Computing degree of parallelism for BPMN processes. In: International conference on service-oriented computing (ICSOC), pp 1–15
44. Liu R, Kumar A (2005) An analysis and taxonomy of unstructured workflows. In: international conference on business process management (BPM), pp 268–284
45. Veanes M, Campbell C, Grieskamp W, Schulte W, Tillmann N, Nachmanson L (2008) Model-based testing of object-oriented reactive systems with Spec Explorer. In: Hierons RM, Bowen JP, Harman M (eds) Formal Methods and Testing. Lecture Notes in Computer Science, vol 4949. Springer, Berlin, Heidelberg
46. Activiti tool documentation. https://github.com/Activiti/activiti-examples. Accessed 19 May 2018
47. Camunda tool documentation. https://camunda.org/examples/. Accessed 19 May 2018
48. Eshuis R, Kumar A (2016) Converting unstructured into semi-structured process models. Data Knowl Eng 101:43–61
49. Polyvyanyy A, García-Bañuelos L, Dumas M (2010) Structuring acyclic process models. In: international conference on business process management (BPM), pp 276–293