Computing and Informatics, Vol. 35, 2016, 719–743

FUNCTIONAL TESTING USING OBJECT WORKFLOW NETS

Stéphane Julia

Computing Faculty – Federal University of Uberlândia
P.O. Box 593, 38400-902
Uberlândia/MG, Brazil
e-mail: stephane@ufu.br

Liliane do Nascimento Vale

Computer Science Department – Federal University of Goiás
Avenida Dr. Lamartine Pinto de Avelar, 1120, 75704-020
Catalão/GO, Brazil
e-mail: liliane.ufg@gmail.com

Lígia Maria Soares Passos

Computer Science Department – Federal Rural University of Rio de Janeiro
Av. Governador Roberto Silveira, s/n, Centro, 26210-210
Nova Iguaçu/RJ, Brazil
e-mail: ligiamaria.soarespassos@gmail.com

Abstract. The main purpose of this paper is to present a new formal definition that can be used for modeling functional test. Initially, WorkFlow nets are used to represent the main functional requirement of the software. Next, Object WorkFlow nets derived from WorkFlow nets and object Petri nets are used to formally specify the test models of object oriented software functionalities to be used. In particular, the proposed models allow for the addition of complex data structure specifications as well as complex control specifications. The dynamic execution of functional testing models, when considering a specific software architecture, is given by the instantiation of a testing class associated with the tested functionality. An example

of execution of functional testing corresponding to the "Withdrawal Operation" of a bank ATM machine is presented, as well as a comparative study based on a more traditional UML modeling approach.

Keywords: Functional testing, object Petri nets, WorkFlow nets

Mathematics Subject Classification 2010: 68-N19, 68-M15

1 INTRODUCTION

Software testing is an essential and complex activity in software engineering. It is essential because it helps to ensure the quality and reliability of the tested software [1]. It is complex mainly due to the actual software complexity. This is a field within the software engineering community which has had many achievements, as shown in [2].

According to Mathur and Malik in [3], software testing is the most important phase in the software development life cycle. So, in the context of software development, it would be essential to include the testing activity among all the steps of software development, with the aim of improving the detection of flaws which may exist as part of it, as proposed in the V-Model, since the purpose of software testing is to determine whether a program contains errors.

Software testing methods can be divided into two broad classes: functional and structural testing. Functional testing is used to find disagreements between the specification and the actual implementation of the software system [4]. This is the activity concerned with the verification of the functionalities of a software and is essentially applied to the phase corresponding to the requirement specification activity which is generally based on modeling techniques. According to Borba et al. in [5], the requirements are one of the primary sources of input to the system test process; so the requirements are also subject to verification. In functional testing, the tester basically considers the specification to obtain test requirements or test data without any concern for implementation. So a high-quality specification is fundamental to support the application of functional testing. The authors in [5] point out that a disadvantage of functional testing is that specifications, mainly informal, may be incomplete or ambiguous, as will be the test suite which is created based on them. To solve this, in recent years, formal methods and software testing are seen as complementary and as two important approaches that assist in the development of high-quality software [6]. According to Hierons et al. in [6], formal specifications and models may be used as the basis for automating parts of the testing process and can lead to more efficient and effective testing. In this context, the authors highlight that the primary idea behind a formal method is the benefit of writing a precise specification for a system. So, a specification of a system might cover its functional as well as its structural or architectural behaviour. By using

formal methods and testing together, it may be capable of generating functional test cases from the specification of a system [6].

Many works have already considered formal methods for functional testing. Different kinds of graphs are used to formalize functional testing techniques for procedural software as shown in [7], but the object-oriented software is not considered. Simão et al. in [8] present the tool Proteum-RS/PN, that explores the criteria of mutation testing in the context of functional testing for Reactive Systems using, in particular, models based on Petri nets. Other tools, with the same feature, were developed based on Finite State Machines [9] and Statecharts [10]. These approaches also consider procedural software. In [11], an approach for functional testing of object-oriented software, based on State Machine specifications is presented. In this approach, the authors are concerned with the functional testing of each method of a class. In [12], a low level Petri net, named Class Petri Net Machine (CPNM), for specifying method sequence specification of a class is proposed, and a test-case generation technique, based on Petri nets, is presented. In this technique, the authors specify the behaviour of classes in terms of CPNM and generate test cases based on a reachability tree that covers the correct sequence usage for all the objects of the analysed class. The focus here therefore also becomes the internal behaviour of a class as an individual unit. In [13], an approach based on the translation of UML 2.0 Activity Diagrams into coloured Petri nets is presented. This approach considers the flow controls, which exist in the UML specification, as well as the data flows represented by objects. Test cases are then applied to the Petri net models representing the internal behaviour of a class. Such an approach does not consider in an explicit way the existing interaction mechanisms between objects. In [1], the authors present the ISTA (Integration and System Test Automation), a tool for the automated generation of an executable test code, based on high-level Petri nets (Predicate/Transition nets or Coloured Petri nets) for specifying test models. The approach presented in this paper is related to the testing of object-oriented software and the ISTA input is a MID (Model-Implementation Description) specification consisting of a Predicate/Transition (PrT) net, a MIM (Model-Implementation Mapping) and HC (Helper Code). In such an approach, more than one model/mapping is necessary to perform a test.

Nowadays, there are many approaches that are focused on test case generation from UML specifications [14, 15, 16]. Many of them consider the business process models as relevant for the testing of the software and are based on UML Activity Diagrams, as shown in [16]. An approach based on business process modeled by BPMN (Business Process Model and Notation) for software testing is shown in [17]. In this approach, the process model contains information about process, flows and tasks, and is taken as an input to generate test-cases that are generated based on these flows. To generate the test-cases, another four models are necessary: architecture, user interface, behaviour and data model.

None of the discussed works presented formal models that integrate data flows as well as control flows and also consider interactions between objects in a formal way. Another limitation which clearly appears in most of the papers which deal

with software testing is the lack of an operational semantic for the test models. As a matter of fact, in order to follow the execution of a test, simulation mechanisms adapted to object-oriented systems should also be associated with the models used to specify the software tests.

The purpose of this paper is to propose the execution and simulation of functional test models that can support parallelism, concurrency, control flow representation and complex data structure specification for object-oriented software.

In Section 2, some theoretical results, necessary in order to define the general test model, are presented. In Section 3, the definition of the test model based on WorkFlow nets and objects Petri nets is given. In Section 4, the implementation principle of the test model using a test class within an object oriented architecture is presented. The dynamic execution and simulation of the test model are explained in Section 5. Section 6 presents a comparative study between the approach presented in this paper and an approach based on UML Diagrams in the context of V-model software development processes. Finally, the conclusions for this paper are given in Section 7.

2 MODELING TECHNIQUES

2.1 Petri Nets

Petri nets are defined as a directed bipartite graph with two types of nodes named places and transitions. The nodes are connected via directed arcs. Connections between two nodes of the same type are not allowed. Places are represented by circles and transitions by rectangles [18, 19]. There are many extensions of Petri net models that can be classified as low-level Petri nets or high-level Petri nets [20].

Low-level Petri nets are characterized by simple tokens in places that indicate the partial states of the system. The definition of a marked Petri net is the following:

Definition 1. A marked Petri net is a 5-tuple $PN = (P, T, F, W, M_0)$ where [19]:

- $P = \{P_1, P_2, \ldots, P_m\}$ is a finite set of places;
- $T = \{T_1, T_2, \ldots, T_n\}$ is a finite set of transitions;
- $F \subseteq (P \times T) \bigcup (T \times P)$ is a set of arcs;
- $W : F \longrightarrow \{1, 2, 3, \ldots\}$ is a weight function;
- $M_0 : P \longrightarrow \mathbb{N}$ is the initial marking.

The global state of the system is then given by the distribution of tokens considering all the places on the net. An example of an ordinary marked Petri net is presented in Figure 1.

A state or marking in a Petri net is transformed into another marking according to the following firing rules:

- A transition t_j is enabled if each input place P_i of the transition is marked with at least $W(P_i, T_j)$ tokens, where $W(P_i, T_j)$ is the weight of the arc from P_i to T_j.

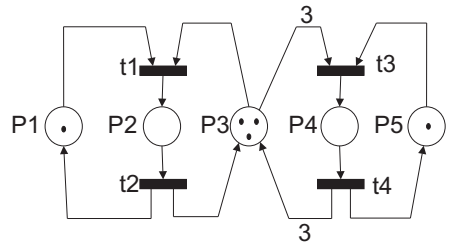

Figure 1. Marked Petri net

- The firing of an enabled transition T_j consumes $W(P_i, T_j)$ tokens from each input place P_i of T_j and produces $W(P_k, T_j)$ tokens for each output place P_k of T_j, where $W(P_k, T_j)$ is the weight of the arc from T_j to P_k. An example of transition firing t_3 is illustrated in Figure 2.

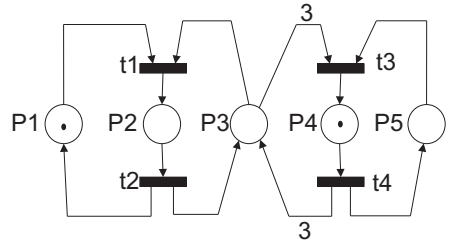

Figure 2. Example of transition firing

Some of the most important properties of Petri nets are the following:

Live Petri net: A marked Petri net is said to be live, if no matter what marking has been reached from M_0, it is possible to ultimately fire any transition of the net by progressing through some further firing sequence.

Bounded Petri net: A marked Petri net is bounded if the number of tokens in each place does not exceed a finite number K for any marking reachable from M_0.

An overview of the Petri net theory can be found in [20].

2.2 WorkFlow Nets

Petri nets can be used as a tool for the representation, validation and verification of Workflow processes [19]. Petri nets which model Workflow processes are called WorkFlow nets. A WorkFlow net respects the following properties [19]:

- It has only one input place, named *Start*, and one output place, named *End*. The place *Start* does not have input arcs and the place *End* does not have output arcs.

- One token in the place *Start* represents a "case" which has to be treated and a token in the place *End* represents a "case" which has been treated.
- The tasks T attached to transitions and the conditions P attached to places must belong to an existing path between the place *Start* and the place *End*.

Figure 3 represents an example of processes for handling complaints which is a kind of Workflow process [19].

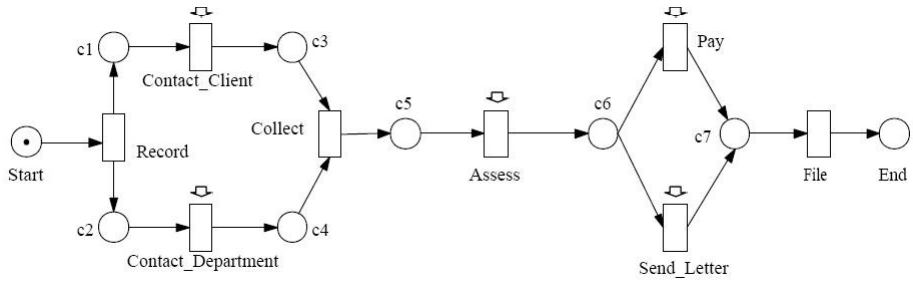

Figure 3. An example of WorkFlow net

Soundness is the main property of WorkFlow nets. A WorkFlow net is *sound* if [19]:

- For each token in the place *Start*, one and only one token is produced in place *End*.
- When the token is produced in place *End*, all the other places are empty for this case.
- For each transition (task), it is possible to move from the initial state to a state in which that transition is enabled, i.e., there does not exist any dead transition.

2.3 Object Petri Nets

Ordinary Petri nets do not allow for the modeling of complex data structures. Many extensions have been proposed to model this specific aspect through high-level Petri net definitions.

The object Petri nets defined by Sibertin-Blanc [21] are based on the integration of predicate/transition Petri nets and the concept of an object oriented paradigm. The tokens are considered as *n-tuples* of instances for a class of object and carries data structures defined as sets of attributes for specific classes. Pre-conditions and actions are associated with transitions, which respectively act on the attributes (eventually modifying their values) of the data structures transported by the tokens of the net. The object Petri nets can be formally defined as:

Definition 2. A marked object Petri net can be defined by the 9-tuple:

$$N_0 = \langle P, T, C_{lass}, V, Pre, Post, A_{tc}, A_{ta}, M_0 \rangle$$

where

- C_{lass} is a finite set of classes of objects: for each class, a set of attributes is also defined;
- P is a finite set of places whose types are given by C_{lass};
- T is a finite set of transitions;
- V is a set of variables whose types are given by C_{lass};
- *Pre* is the function precedent place (an arc between a place and a transition which considers a formal sum of elements of V);
- *Post* is the function next place (an arc between a transition and a place which considers a formal sum of elements of V);
- A_{tc} is an application which associates to each transition a condition that involves the attributes of the formal variables associated with the input arcs of the transitions;
- A_{ta} is an application which associates to each transition an action that involves the formal attributes of the variables associated with the input arcs of the transitions and updates the attributes of the formal variables associated with the output arcs of the transitions;
- M_0 is the initial marking which associates a formal sum of objects to each place (n-tuples of instances of classes that belong to C_{lass});

An example of object Petri net is presented in Figure 4. The set of classes is defined as:

$$C_{lass} = \{Product, Request\}.$$

The *Product* class has the attributes:

$$\begin{cases} name = identifier; \\ code = integer; \\ cost = float; \end{cases}$$

The *Request* class has the attributes:

$$\begin{cases} code : integer; \\ cost : float; \\ type : identifier; \end{cases}$$

The variable *pr* belongs to the *Product* class and the variable *pd* belongs to the *Request* class. The place *Products Stock* belongs to the *Product* class, the place *Buffer Request* belongs to the *Request* class and the place *Processed Requests* belongs to the *Request* class. The initial marking M_0 is given by the objects that are in the places *Poducts Stock* and *Buffer Request*:

$$M_0 = \begin{bmatrix} \langle pr1 \rangle + \langle pr2 \rangle + \langle pr3 \rangle \\ \langle pd1 \rangle + \langle pd2 \rangle \\ 0 \end{bmatrix}.$$

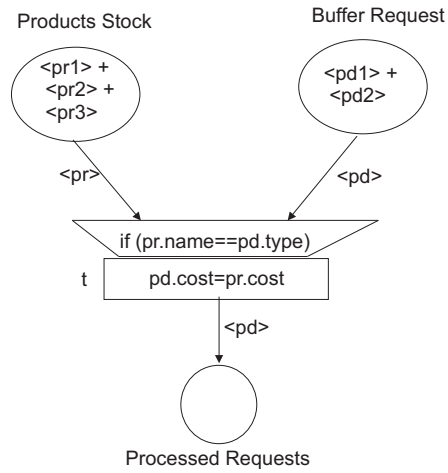

Figure 4. Specification of a sale transaction

For example, the attributes of the object (token) $pr1$ can be given by:
Product $pr1$;
name : *home theater*;
code : 567544;
cost : $278, 50$;
and the attributes of the object (token) $pd2$ can be given by:
Request $pd2$;
code : 123440;
cost : $00, 00$;
type: home theater;

The detailed definition of the dynamic behaviour (firing rules) of the object Petri nets can be found in [21]. In Figure 4, the transition t is enabled by the initial marking. The attributes of the variable pr associated with the arc connecting the place *Products Stock* to the transition t can be replaced by the attributes of the objects $pr1$ for example. Similarly, the attributes of the variable pd associated with the arc connecting the place *Buffer Request* to transition t can be replaced by the attributes of the objects $pd2$ for example. Considering that the attributes of the pair of objects $(pr1, pd2)$ check the condition associated with the transition t, the transition can be fired. The action associated with the transition is then executed and a new object $pd2$ can be produced in the place *Processed Requests*, as shown in Figure 5, with the following attributes:
Request $pd2$;
code : 123440;
cost : $278, 50$;
type : *home theater*;

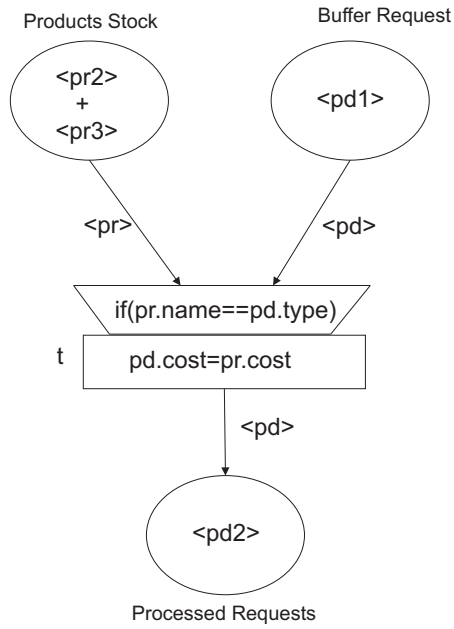

Figure 5. Execution of a sale transaction simulation

In particular, when considering this new object *pd*2, the attribute *cost* has been modified after the firing of t.

3 FUNCTIONAL TESTING SPECIFICATION MODEL

The implementation of a test model for oriented object software can be seen as the execution of a test case of a Workflow process. A test case has a beginning, an end, and performs various operations (it tests several methods of the called objects in order to execute a given functionality) following different types of routings such as: sequential routing, alternative routing, parallel routing or iterative routing.

A test case should also allow for the specification of complex data structures and be a sufficiently formal model so as to be implemented and easily transformed into an executable code. The definition of the specification model for the functional testing of object oriented software is then given by the following definition:

Definition 3. The specification model for the functional testing of object oriented software is defined by an object Petri net in such a way that the underlying autonomous Petri net used to define the control structure of the test is given by a WorkFlow net. The corresponding object WorkFlow net used as the final test model is then defined by the 9-tuple:

$$N_0 = \langle T_{estClass}, P, T, V, Pre, Post, A_{tc}, A_{ta}, M_0 \rangle$$

where

- $T_{estClass}$ represents the test type that is defined by a set of attributes of two types:
  - attributes that represent the input data of the test,
  - attributes that represent the output data of the test.
- P is a finite set of places, which in particular possesses the places *Start* and *End* which are the underlying WorkFlow net places. The element types for P are those defined in $T_{estClass}$.
- T is a finite set of transitions.
- V is a set of variables whose types are given by $T_{estClass}$.
- *Pre* is the function precedent place which considers a formal sum of elements of V.
- *Post* is the function next place which considers a formal sum of elements of V.
- A_{tc} is an application which associates to each transition a condition that involves the attributes of the formal variables associated with the input arcs of the transitions.
- A_{ta} is an application which associates to each transition an action that involves the formal attributes of the variables associated with the input arcs of the transition and updates the attributes of the formal variables associated with the output arcs of the transitions.
- M_0 is the initial marking which associates to the place *Start* a test object whose attributes are instantiated in $T_{estClass}$.

To illustrate the main features of the given definition, an example of functionality of an ATM (Automated Teller Machine) is considered. The informal specification of the studied functionality is the following: *Initially, the customer inserts the identification card. If the identification card is valid, then the customer enters the password. If the password is correct, the customer executes an operation of drawing. If the balance in the customer's account is sufficient, the customer receives the money.*

The functionality specified in natural language does not formally define what is exactly expected from the drawing function. The different interpretations of the textual specification can produce several control specifications given by several WorkFlow net models. The sequential case is illustrated by the WorkFlow net in Figure 6. In this model, all the operations are executed in a perfect sequence. The charge-amount operation updates the balance of the client after the drawing and the safe-update operation updates the amount of money available in the ATM machine.

The iterative case is illustrated by the WorkFlow net of Figure 7. If the user enters a wrong password, this model allows various attempts to enter the correct password, featuring an iterative route in the corresponding WorkFlow net.

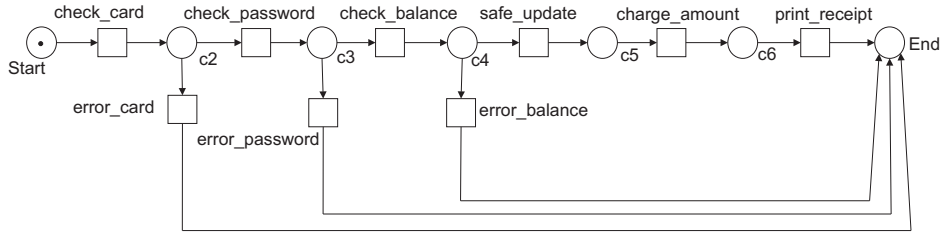

Figure 6. WF-Net: Representation of generic ATM system by sequential WorkFlow net

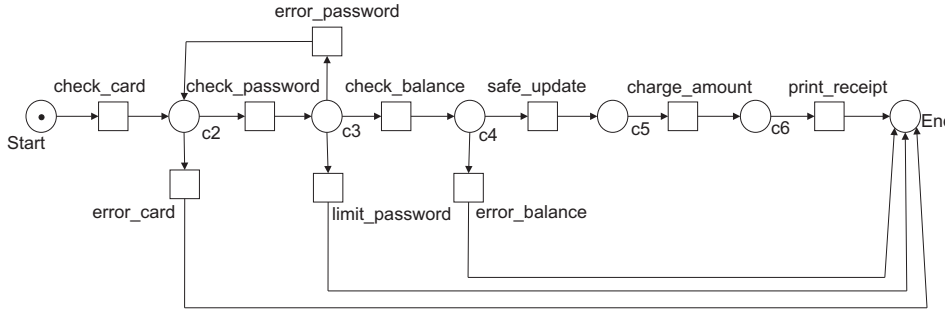

Figure 7. WF-Net: ATM iterative system representation by WorkFlow net

Parallelism can be represented in the control specification of the functional test, as illustrated by the WorkFlow net in Figure 8. In this example, the WorkFlow net indicates that the order of execution of operations "print-receipt", "charge-amount" and "safe-update" is not important in the test execution and can be implemented in parallel or in any sequence if the system does not allow true parallelism.

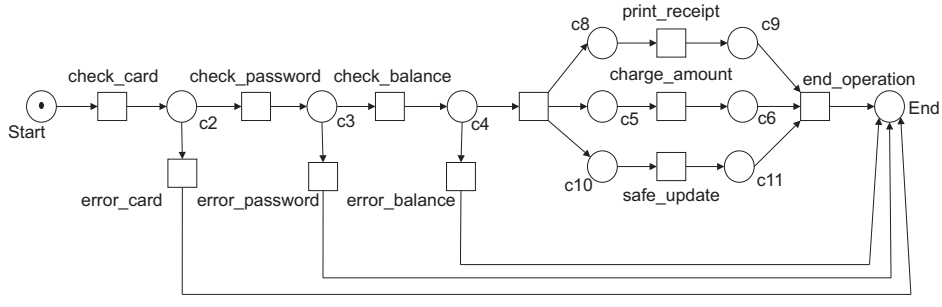

Figure 8. WF-Net: ATM parallel system representation by WorkFlow net

Once the control structure of the functional test is specified by an ordinary WorkFlow net, the final model of the test can be produced using the definition of an object WorkFlow net. For example, if the control structure of the test is the one

specified by the WorkFlow net in Figure 6 (the sequential case), then the functional test model will be given by the object WorkFlow net in Figure 9. In this model, the transitions are associated with methods (operations) that manipulate the attributes provided by the test object cl_1 which is initially in the place *Start*. cl_1 is an instance of the class *TestClass* defined as:

TestClass;
Input Test Data:
number*: integer; // card number*
password*: integer; //client password*
drawing*: float; //amount of money*

Output Test Data:
nbr*: Boolean; //indicates if the card number is genuine*
s*: Boolean; // indicates if the password is correct*
bal*: Boolean; // indicates if the amount of money indicated by the client is acceptable according to the amount of money that exists in his bank account*
imp*: Boolean; // indicates if a receipt of the transaction has been printed*
n*: Boolean; // indicates if the amount of money of the ATM machine has been actualized after the transaction*
balance*: float; // indicates the balance of the client after the drawing operation*

The operations associated with the transitions are:

- *getCheckCard*: checks the validity of the card, manipulating the attribute *number* that provides the code for the card to be analyzed. The result is then stored in the Boolean variable *nbr* indicating if the card is valid or not.
- *getCheckPassword*: checks the validity of the password, manipulating the attribute *password* that provides the password of the card to be analyzed. The result is then stored in the Boolean variable s indicating if the password is valid or not.
- *getCheckBalance*: checks if there is a balance, manipulating the attribute *drawing* that provides the value of drawing required to be analyzed. The result is then stored in the Boolean variable *bal* indicating if the balance is sufficient or not.
- *getSafeUpdate*: updates the value of the safe after the drawing is completed, manipulating the attribute *drawing*. The result is then stored in the Boolean variable n, indicating if the safe has been updated or not.
- *getDebitAmount*: updates the value of the balance of the client after the drawing is completed, manipulating the attribute *drawing*. The result is then stored in the float variable *balance*, indicating the balance of the client.
- *getPrintReceipt*: prints a receipt for the customer, informing the transaction made through the attributes *drawing* and *number*. The result is then stored in

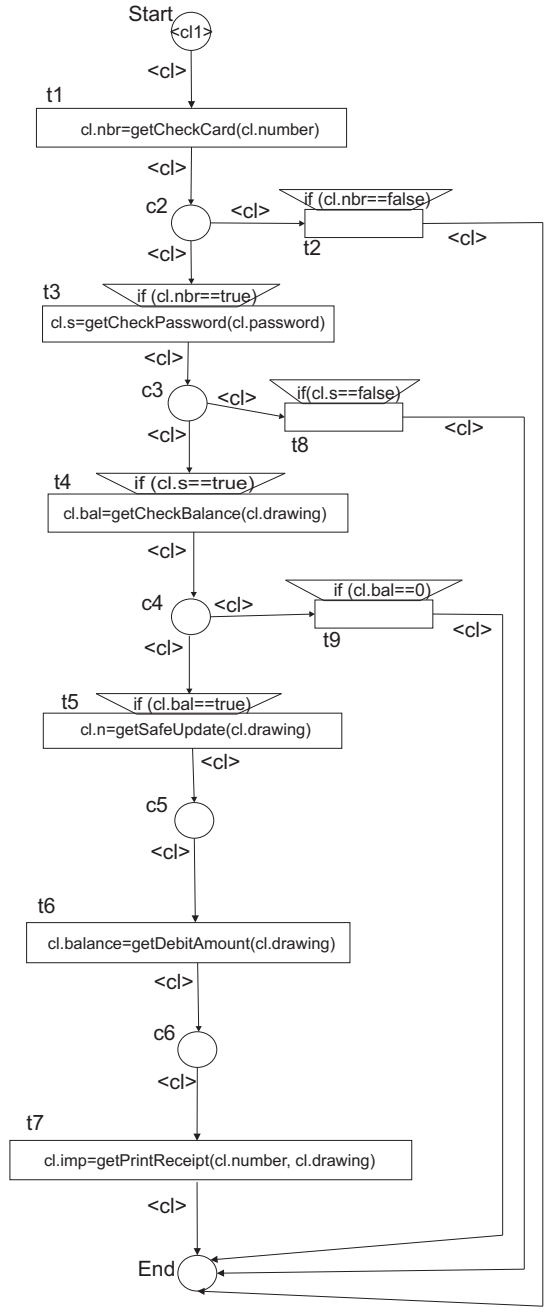

Figure 9. Object WF-Net: Functional testing specification model for generic ATM system in the sequential case

the Boolean variable *imp*, indicating whether the receipt was printed correctly or not.

It is important to note that some transitions have pre-conditions that must be satisfied to allow a transition firing. For example, if the condition $if(cl.nbr == true)$ of the transition t_3 is true then the password can be tested; otherwise it means that the condition $if(cl.nbr == false)$ is true and the transition t_2 is fired, producing a final object in place *End*.

4 IMPLEMENTATION OF THE FUNCTIONAL TESTING SPECIFICATION MODEL

The implementation of the functional testing specification model corresponds to the instantiation of a generic testing class whose main method is the unmarked object WorkFlow net corresponding to the tested functionality. As a matter of fact, the marking of the object WorkFlow net will represent the creation of a specific object used for a specific test.

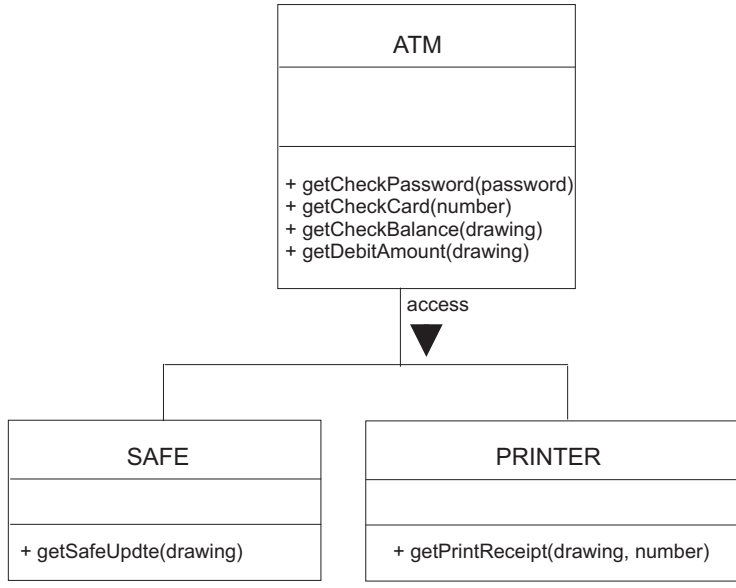

Figure 10. Class diagram involving the main ATM generic system classes

When applying the functional testing activity, the architecture of the software is already known. In the context of the example of the drawing function, part of the software architecture can be given by the class diagram of Figure 10. The testing class will then have to interact with the main classes of the software architecture in order to test the set of methods involved in the tested functionality, as shown in Figure 11.

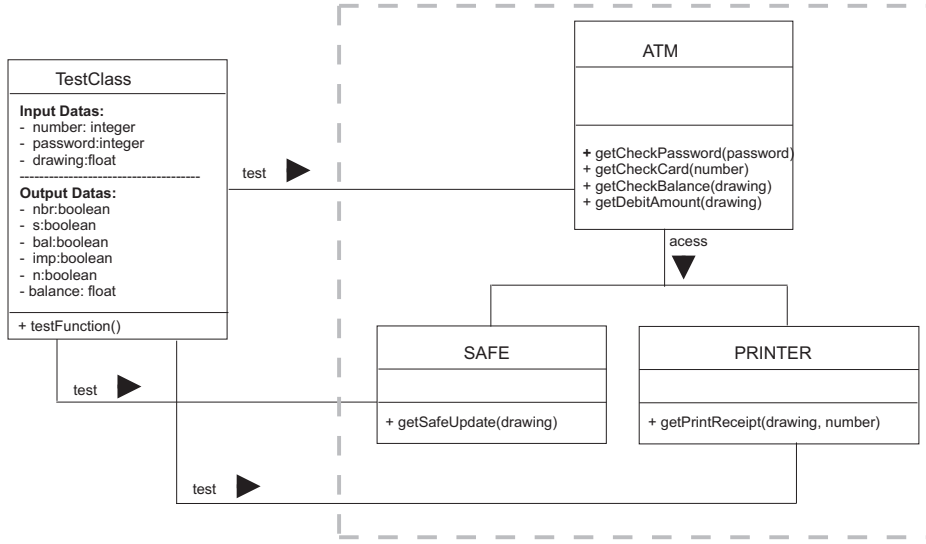

Figure 11. Interaction between the testing class and the main ATM generic system classes

Figure 12 illustrates the main method of the test class in the sequential case. The methods that are called through the transitions of the object WorkFlow net are those tested during the functional testing execution.

5 TEST SCENARIO EXECUTION

In this section, a test scenario execution that considers a specific test case is presented. The control structure of the test is that given by the object WorkFlow net of Figure 12.

The initial marking of the net is then given by a testing object (token) cl_1 in the initial place *Start* which represents the test case.

The attribute values of the object cl_1 at the beginning of this test execution are:

cl_1 ***TestClass***;
Input Test Data:
number: 123456;
password: 5996084;
drawing: 200,00;

Output Test Data:
nbr : false;
s : false;
bal : false;
imp : false;
n : false;
balance : 1000;

For such values, the test execution will be given through the following sequence of steps:

[**Transition t1:** The transition t1 calls the **getCheckCard** method of the **atm** object from the **ATM** class, and verifies if the card number is valid. The result is then stored in the boolean variable **nrb**]

[**Transition t2:** if the pre-condition **if(cl.nbr==false)** is satisfied, the card has presented identification problems. The transition t2 ends the test]

[**Transition t3:** if the pre-condition **if(cl.nbr==true)** is satisfied, the card **number** has not presented identification problems. The transition t3 calls the **getCheckPassword** method of the **atm** object from the **ATM** class, and verifies if the password given by the owner of the card is the same as the one stored in the card. The result is stored in the boolean variable **s]**

[**Transition t4:** if the pre-condition **if(cl.s=true)** is satisfied, the **password** provided by the customer is correct. The transition t4 calls the **getCheckBalance** of the **atm** object from the **ATM** class and verifies if the balance is sufficient. The result is stored in the boolean variable **bal**]

[Transition t5: if the pre-condition **if(cl.bal=true)** is satisfied, the customer balance is positive. The transition t5 calls the method **getSafeUpdate** of the **safe** object from the **Safe** class and verifies if the safe was updated after the customer drawing and the result is stored in the boolean variable **n**]

[**Transition t6**: the transition t6 calls the **getDebitAmount** method of the **atm** object from the **ATM** class and charges on the customer account the drawed amount. The result is stored in the float variable **balance]**

[**Transition t7:** the transition t7 calls the **getPrintReceipt** method of the **print** object from the **Printer** Class and verifies if the printer can print the customer receipt. The result is stored in the boolean variable **imp**]

[**Transition t8:** if the pre-condition **if(cl.s==false)** is satisfied, the password provided by the user is wrong. The transition t8 ends the test]

[**Transition t9**: if the pre-condition **if(cl.bal==false)** is satisfied, the customer balance is negative. The transition t9 ends the test.]

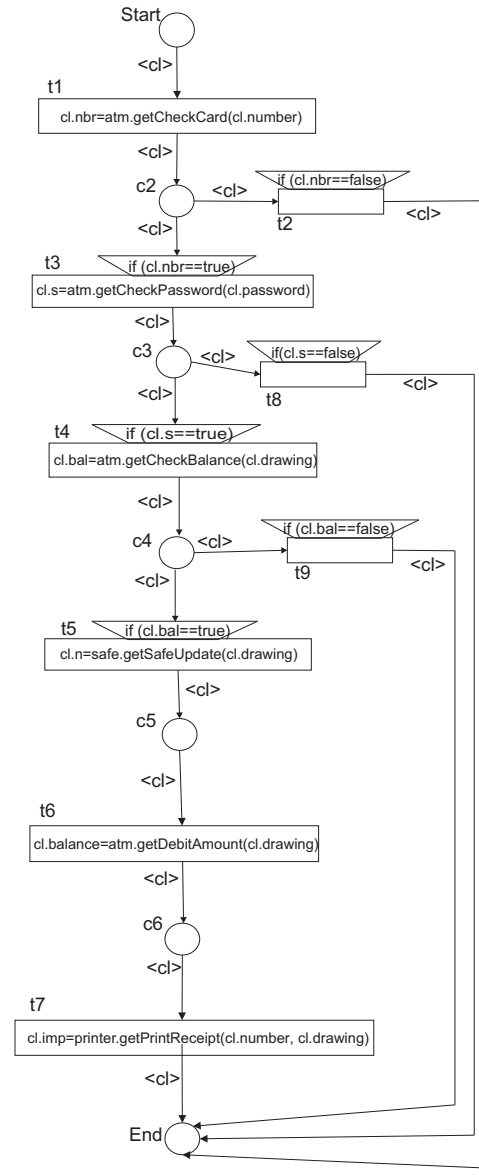

Figure 12. Object WorkFlow net: Method "TestFunction()" of the "*TestClass*"

- The initial marking (object cl_1 in place start) enables the transition t_1, which has no pre-condition. The firing of t_1 calls the method *getCheckCard* of the object *atm* from the *ATM* class, passing, in particular, the attribute *number* = *123456* (number on the card). The called method verifies the existence of the card and returns a Boolean value saved in the attribute *nbr* of the object cl_1. After the firing of t_1, a new object cl_1 (with the attribute *nbr* = *true* modified at the end of the firing and indicating that the card number is available) is produced in place c_2.
- With an object cl_1 in c_2, the transitions t_2 and t_3 are enabled. The pre-condition associated with the transition t_3 which specifies the existence of the card through the attribute *nbr* is satisfied. The firing of t_3 then calls the method *getCheck-Password* of the object *atm* from the *ATM* class, passing as a parameter the attribute *password* = 599604 (password of the cards). The called method verifies the validity of the password and returns a Boolean value stored in the attribute s of the object cl_1 . After the firing of t_3, a new object cl_1 (with the attribute s = *true* modified at the end of the firing and indicating that the password is correct) is produced in place c_3.
- With an object cl_1 in c_3, the transitions t_4 and t_8 are enabled. The pre-condition associated with the transition t_4 which specifies the validity of the password through the attributes is satisfied. The firing of t_4 then calls the method *getCheckBalance* of the object *atm* from the *ATM* class, passing as a parameter the attribute *drawing* = 200.00 (amount to be drawn by the client). The called method verifies the validity of the withdrawal requested by the client and returns a Boolean value stored in the attribute *bal* of the object cl_1. After the firing of t_4, a new object cl_1 (with the attribute *bal* = *true* modified at the end of the firing and indicating that the balance of the customer is sufficient) is produced in place c_4.
- With an object cl_1 in c_4, the transitions t_5 and t_9 are enabled. The pre-condition associated with the transition t_5 which specifies the existence of a positive balance through the attribute *bal* is satisfied. The firing of t_5 then calls the method *getSafeUpdate* of the object *safe* from the *Safe* class, passing as a parameter the attribute *drawing* = 200.00 (amount to be drawn by the client). The called method updates the balance of the ATM machine considering the withdrawal requested by the client and returns a Boolean value stored in the attribute indicating that the balance of the ATM machine was updated. After the firing of t_5, a new object cl_1 (with the attribute n = *true* modified at the end of the firing and indicating that the value was updated) is produced in place c_5.
- With an object cl_1 in c_5, the transition t_6 is enabled. The firing of t_6 then calls the method *getDebitAmount* of the object *atm* from the *ATM* class , passing as a parameter the attribute *drawing* = 200.00 (amount to be drawn by the client). The called method updates the balance of the client considering the withdrawal requested by the client and returns the clients balance. After the

firing of t_6, a new object cl_1 (with the attribute *balance* = 800 modified at the end of the firing and indicating the value of the client's balance) is produced in place c_6.

- With an object cl_1 in c_6 the transition t_7 is enabled. The firing of t_7 then calls the method *getPrintReceipt* of the object *impr* from the *Printer* class, passing as a parameter the attributes *drawing* = 200.00 (amount drawn by the client) and *number* = 123456 (number on the card). The called method verifies that the printer is available, returning a Boolean value stored in the attribute *imp* of the object cl_1. After the firing of t_7, a new object cl_1 (with the attribute $imp = true$ modified at the end of the firing and indicating that the printer is available) is produced in place *End*.

Initial Value	After Firing	Final Value
$cl_1.nbr = false$	t_1	$cl_1.nbr = true$
$cl_1.s = false$	t_3	$cl_1.s = true$
$cl_1.bal = false$	t_4	$cl_1.bal = true$
$cl_1.n = false$	t_5	$cl_1.n = true$
$cl_1.balance = 1000$	t_6	$cl_1.balance = 800$
$cl_1.imp = false$	t_7	$cl_1.imp = true$

Table 1. Values of attributes after the scenario execution

At the end of the test execution, Table 1 is obtained. Such a table shows the final values of attribute of the object cl_1 at the end of the test. One observes in particular that the card number ($nbr = true$) was accepted by the system, the password was correctly entered ($s = true$), the withdrawal was made considering positive balance ($bal = true$), the cashier was updated ($n = true$) after the withdrawal, the customer's bank balance was updated after the withdrawal (*balance* = 800) and, finally, the printer correctly printed an informative message on the screen ($imp = true$).

6 A COMPARATIVE STUDY

This section presents a comparative study between the approach presented in this paper and an approach based on UML diagrams as that presented in [22]. The V-Model used in software development [23] will be considered to associate each diagram produced in functional testing to a specific activity of the software life cycle.

According to Pressman in [23] and Mathur and Malik in [3], the traditional V-Model can be seen as an extension of the waterfall model. Its specificity is due to the fact that each phase of the software development process is associated with specific stages of testing activities. In particular, one of the initial phases that corresponds to requirement specification (Stage 1) is related to a specific phase of elaboration of functional testing models (Stage 2). The functional testing models produced in Stage 2 are then executed in Stage 3 in order to validate the requirement models produced in Stage 1 as presented in Figure 13.

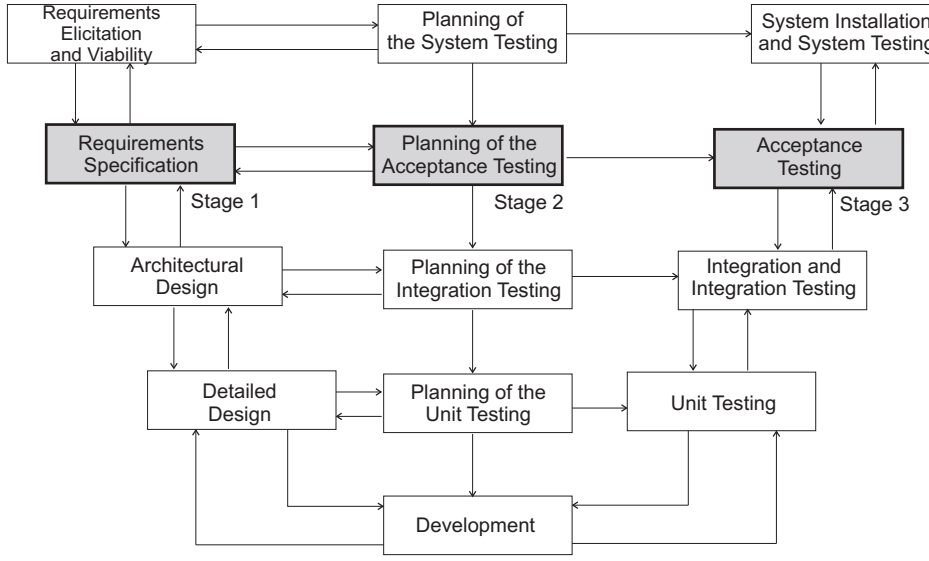

Figure 13. V-Model

Following the approach presented in [22], three kinds of UML Diagrams can be produced that will be associated to the three different stages 1, 2 and 3 of the V-Model. At the requirement analysis (Stage 1), an Activity Diagram should be designed. Activity Diagrams can represent the activities that are carried out by a system in a procedural form. They are well adapted when the main objects of the software architecture are not yet defined, as it is generally the case at the beginning of the software development process. Considering the ATM system presented in the previous sections, the Activity Diagram in Figure 14 is produced.

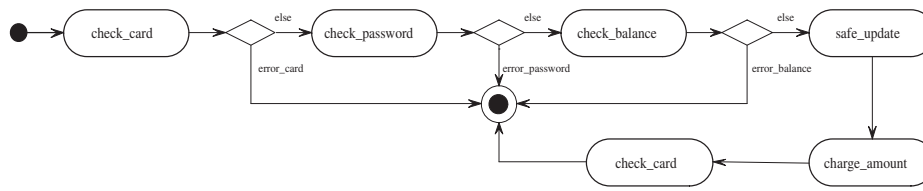

Figure 14. Activity Diagram

Such a diagram shows in a semi-formal way the activities that are carried out by the ATM system in the sequential case. The corresponding model in the approach presented in this paper is the WorkFlow net found in Figure 6 which can be seen as the formal specification of the Activity Diagram in Figure 14. As a matter of fact, the operational semantic of Activity Diagrams is derived from Petri nets in the UML Metamodel [24]. The main advantage in working with a WorkFlow net

model instead of an Activity Diagram is the formal operational semantic which allows for the true simulation through the corresponding token player algorithm and the possibility of verifying formally the Soundness property which proves the correctness of the requirement specification. At Stage 2, where the first functional testing models are produced, System Sequence Diagrams that describe the different scenarios to be tested can be produced, as that shown in Figure 15.

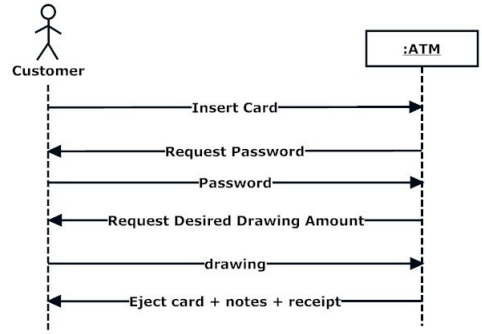

Figure 15. System Sequence Diagram

Such a diagram shows the interactions between a user and the ATM system in order to respect the specification given by the Activity Diagram in Figure 14. At Stage 1 of the development process, the main objects of the future software architecture are not known and interactions can only be specified with a global object called "System". Such a diagram allows for the representation of the called activities as well as basic data flows but does not have a true operational semantic which allows for a kind of scenario validation through simulation. The corresponding model produced in Stage 2 of the approach presented in this paper is the object WorkFlow presented net in Figure 9 which allows true data processing as well as formal simulation. Due to the fact that the control structure is the same as that found in the WorkFlow net shown in Figure 6, the Soundness property of the underlying control structure of the model will continue to be respected and the initial procedural specification will also be maintained.

At Stage 3, the objects of the software architecture are known and the System Sequence Diagram produced in Stage 2 can be transformed into a Sequence Diagram where the objects involved in the specified scenario are represented. Considering the Class Diagram in Figure 10 and the System Sequence Diagram in Figure 15, the Sequence Diagram in Figure 16 is produced and corresponds to the scenario to be tested in order to verify the requirement specification given by the Activity Diagram in Figure 14.

The model that corresponds to the functional testing activity of Stage 3 is the object WorkFlow net presented in Figure 12. The difference between this model and the one used in Stage 2 is simply the fact that at Stage 3 the objects of the

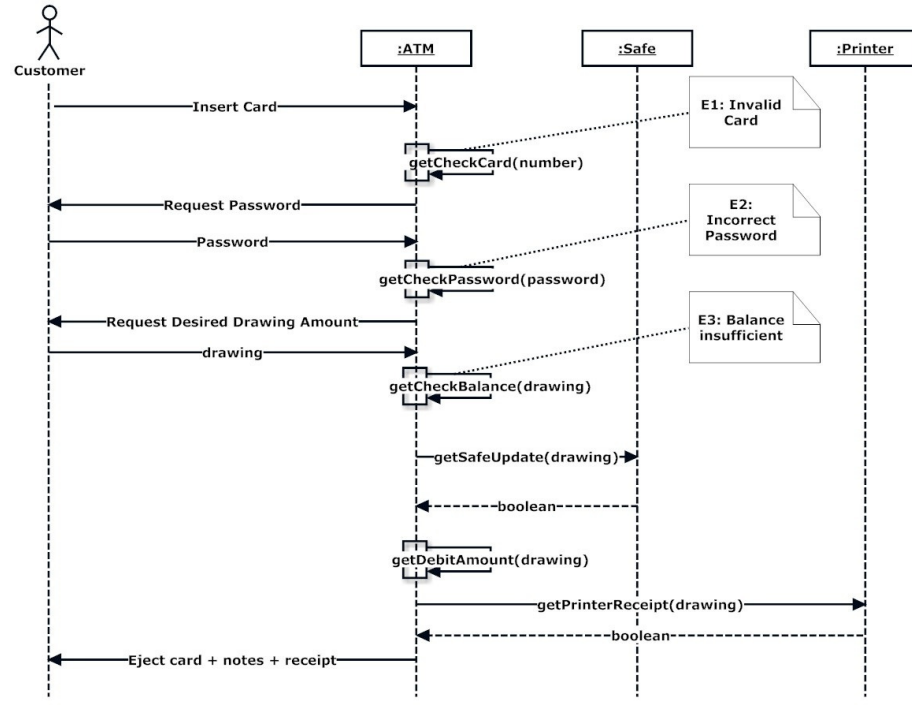

Figure 16. Sequence Diagram

architecture are known and their names can be associated with the called method associated with the firing of the transitions of the corresponding object WorkFlow net.

One notes that the approach presented in this paper has a formal operational semantic allowing for true simulation. In particular, the procedural requirement model is maintained through the different stages of the V-Model and true data processing specification is also allowed by the functional testing model. When a pure UML approach is used to follow the same steps in the functional testing activities, different diagrams have to be used. The Activity Diagram allows a kind of procedural specification but does not allow for the representation of the different interactions between the objects of the software architecture. Sequence Diagrams allow for interaction between objects as well as basic data flow representation but do not keep the procedural form of the initial requirement Activity Diagram and do not possess the real simulation rules necessary to execute the specified scenario of the corresponding functional test.

Finally, working with an object WorkFlow net instead of a UML Diagram, the final model obtained in Stage 3 can be easily transformed and implemented as shown in Figure 11 into a testing class whose instantiations will correspond to the different

sets of data that will be tested when considering the same scenario several times, as it is generally the case with functional testing. A Sequence Diagram will only guide the functional test in Stage 3 but will not truly implement the scenario to be tested.

7 CONCLUSIONS

In this paper, a formal approach based on WorkFlow nets and object Petri nets for the modeling of functional testing was proposed. The main goal was to offer an alternative to the usual informality in which the functional testing activity happens until the present moment.

Petri nets showed themselves to be an appropriate tool for providing a formal definition for a functional testing model. As a matter of fact, the features of functional testing models are very similar to those which exist in workflow processes. Both model types follow the same life cycle with a beginning, the execution of several operations in sequence or not, and an end. In this contest, the WorkFlow nets as a model for the control structure of functional tests were chosen. The capacity of object Petri nets in representing true data structures favoured the possibility of formally specified attributes existing in the functional testing activity.

As a future paper proposal, it would be interesting to implement a compiler or interpreter that automatically generates the test class from the formal specification model. It would also be important to apply the test model defined in this article to one of the several methods used in functional testing such as ISVV (Independent Software Validation Verification) [25], for example.

Acknowledgements

The authors would like to thank CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior), CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico – Brazil) and FAPEMIG (Fundação de Amparo a Pesquisa do Estado de Minas Gerais – Brazil) for their partial financial support to this research. The authors also would like to thank the anonymous referees for their valuable comments.

REFERENCES

[1] Xu D.: A Tool for Automated Test Code Generation from High-Level Petri Nets. Proceedings of the 32nd International Conference on Applications and Theory of Petri Nets (Petri Nets'11). Springer-Verlag, Berlin, Heidelberg, 2011, pp. 308–317. http://dl.acm.org/citation.cfm?id=2022192.2022212.

[2] Bertolino A.: Software Testing Research: Achievements, Challenges, Dreams. Future of Software Engineering (FOSE'07), 2007, pp. 85–103, doi: 10.1109/FOSE.2007.25.

[3] Mathur, S.—Malik, S.: Advancements in the V-Model. International Journal of Computer Applications, Vol. 1, 2010, No. 12, pp. 29–34.

[4] Omar, A. A.—Mohammed, F. A.: A Survey of Software Functional Testing Methods. SIGSOFT Software Engineering Notes, Vol. 16, 1991, No. 2, pp. 75–82, doi: 10.1145/122538.122551. http://doi.acm.org/10.1145/122538.122551.

[5] Borba, P.—Cavalcanti, A.—Sampaio, A.—Woodcock, J. (Eds.): Testing Techniques in Software Engineering. Springer-Verlag, Berlin, Heidelberg, 2010.

[6] Hierons, R. M.—Bogdanov, K.—Bowen, J. P.—Cleaveland, R.—Derrick, J.—Dick, J.—Gheorghe, M.—Harman, M.—Kapoor, K.—Krause, P. et al.: Using Formal Specifications to Support Testing. ACM Computing Surveys (CSUR), Vol. 41, 2009, No. 2, Art. No. 9, doi: 10.1145/1459352.1459354. http://doi.acm.org/10.1145/1459352.1459354.

[7] Beizer, B.: Black-Box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[8] Simão, A. S.—Maldonado, J. C.—Fabbri, S. C. P. F.: Proteum-RS/PN: A Tool to Support Edition, Simulation and Validation of Petri Nets Based on Mutation Testing. XIV Brazilian Symposium on Software Engineering (SBES 2000), 2000.

[9] Maldonado, J. C.—Delamaro, M. E.—Fabbri, S. C. P. F.—Simão, A. S.—Sugeta, T.—Vincenzi, A. M. R.—Masiero, P. C.: Mutation Testing for the new Century. Chapter Proteum: A Family of Tools to Support Specification and Program Testing Based on Mutation. Kluwer Academic Publishers, Norwell, MA, USA, 2001, pp. 113–116. http://dl.acm.org/citation.cfm?id=571305.571329.

[10] Sugeta, T.—Maldonado, J. C.—Masiero, P. C.—Fabbri, S. C. P. F.: Proteum/st: A Tool to Support Statecharts Validation Based on Mutation Testing. Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes de Software (Ideas 2001), 2001, pp. 370–384.

[11] Georgieva, J.—Gancheva, V.: Functional Testing of Object-Oriented Software. Proceedings of the 4th International Conference on Computer Systems and Technologies: E-Learning (CompSysTech '03). ACM, New York, NY, USA, 2003, pp. 141–146, doi: 10.1145/973620.973644. http://doi.acm.org/10.1145/973620.973644.

[12] Wang, C. C.—Pai, W.—Chiang, D. J.: Using a Petri Net Model Approach to Object-Oriented Class Testing. 1999 IEEE International Conference on Systems, Man, and Cybernetics (SMC '99), Vol. 1, 1999, pp. 824–828, doi: 10.1109/ICSMC.1999.814198.

[13] Chen, M.—Qiu, X.—Li, X.: Automatic Test Case Generation for UML Activity Diagrams. In: Zhu, H., Horgan, J. R., Cheung, S. C., Li, J. J. (Eds.): Proceedings of the 2006 International Workshop on Automation of Software Test (AST '06). ACM, 2006, pp. 2–8.

[14] Offutt, A. J.—Liu, S.—Abdurazik, A.—Ammann, P.: Generating Test Data from State-Based Specifications. Software Testing, Verification and Reliability, Vol. 13, 2003, No. 1, pp. 25–53.

[15] Andrews, A. A.—France, R. B.—Ghosh, S.—Craig, G.: Test Adequacy Criteria for UML Design Models. Software Testing, Verification and Reliability, Vol. 13, 2003, No. 2, pp. 95–127.

[16] Kaur, A.—Vig, V.: Systematic Review of Automatic Test Case Generation by UML Diagrams. International Journal of Engineering Research and Technology, Vol. 1, 2012, No. 6.

[17] Gupta, P.—Surve, P.: Model Based Approach to Assist Test Case Creation, Execution, and Maintenance for Test Automation. Proceedings of the First International Workshop on End-to-End Test Script Engineering (ETSE '11). ACM, New York, NY, USA, 2011, pp. 1–7, doi: 10.1145/2002931.2002932. http://doi.acm.org/10.1145/2002931.2002932.

[18] van der Aalst, W. M. P.: The Application of Petri Nets to Workflow Management. Journal of Circuits, Systems, and Computers, Vol. 8, 1998, No. 1, pp. 21–66.

[19] van der Aalst, W. M. P.—van Hee, K.: Workflow Management: Models, Methods, and Systems. The MIT Press, 2004.

[20] Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE, Vol. 77, 1989, No. 4, pp. 541–580.

[21] Sibertin-Blanc, C.: High Level Petri Nets with Data Structure. In: Jensen, K. (Ed.): Proceedings of the 6th European Workshop on Application and Theory of Petri Nets, Espoo, Finland, 1985, pp. 141–170.

[22] Roques, P.: UML in Practice: The Art of Modeling Software Systems Demonstrated Through Worked Examples and Solutions. John Wiley & Sons, 2004.

[23] Pressman, R.: Software Engineering: A Practitioner's Approach. 7th Edition. McGraw-Hill, Inc., New York, NY, USA, 2010.

[24] OMG. OMG Unified Modeling Language Specification – Version 2.4.1. Object Management Group 2011. http://www.omg.org/spec/UML/2.4.1/Superstructure.

[25] Ambrósio, A. M.—Mattiello-Francisco, M. F.—Martins, E.: An Independent Software Verification and Validation Process for Space Applications. Proceedings of the 9th Conference on Space Operations (SpaceOps 2008), AIAA, 2008.

Stéphane Julia received his Ph.D. degree from the Paul Sabatier University of Toulouse, France, in 1997. He is currently Professor at the Computation Faculty of the Federal University of Uberlândia, Brazil. His current research interests include the application of Petri net theory in workflow management systems and software engineering.

Liliane do Nascimento Vale received her Master's degree in computer science from the Federal University of Uberlândia, Brazil, in 2009. She is currently a Computer Science D.Sc. student also at Federal University of Uberlândia and Professor at Federal University of Goiás, Brazil. She has experience in computer science, with the emphasis on software engineering, working lines of Petri nets, testing software, software reuse and recovery software architecture.

Lígia Maria Soares Passos received her D.Sc. degree in computer science from the Federal University of Uberlândia, Brazil, in 2016. She is currently Professor at the Computer Science Department of the Federal Rural University of Rio de Janeiro, Brazil. Her current research interests include the application of Petri net theory in workflow management systems and software engineering.