INFORMATYKA
CZASOPISMO TECHNICZNE
TECHNICAL TRANSACTION
WYDAWNICTWO POLITECHNIKI KRAKOWSKIEJ
COMPUTER SCIENCE AND INFORMATION SYSTEMS
1-I/2011
ZESZYT 9
ROK 108
ISSUE 9
YEAR 108

KRZYSZTOF SAPIECHA, JOANNA STRUG[*]

AN APPLICATION OF UML MODELS FOR SEMI-AUTOMATED GENERATION OF TEST SCENARIOS FOR VALIDATION OF OBJECT ORIENTED SYSTEMS AGAINST SPECIFICATION REQUIREMENTS

ZASTOSOWANIE MODELI UML DO GENERACJI SCENARIUSZY TESTOWYCH DO WALIDACJI SYSTEMÓW OBIEKTOWYCH WZGLĘDEM SPECYFIKACJI WYMAGAŃ

A b s t r a c t

Systematic and rigorous validation carried out during entire process of development of a software intensive system can significantly reduce total costs and time to market and improve the quality of the system. In context of developing object-oriented systems (OOSs) information relevant to validation can be derived from early UML models. The paper presents next results of the work on a novel approach to generation of test scenarios from UML models for validation of OOSs against specification requirements.

Keywords: validation, UML, object-oriented system, test scenarios

S t r e s z c z e n i e

Systematyczna walidacja, prowadzona w trakcie projektowania systemu informatycznego, może w znaczący sposób zredukować czas oczekiwania na produkt i poprawić jego jakość. W kontekście projektowania systemów obiektowych, informacje istotne dla walidacji mogą być uzyskane na postawie modeli UML. W tym artykule zaprezentowano kontynuację prac nad nową metodą generacji scenariuszy testowych do walidacji systemów obiektowych na podstawie modeli UML.

Słowa kluczowe: walidacja, UML, system obiektowy, scenariusze testowe

[*] Prof. dr hab. inż. Krzysztof Sapiecha, dr inż. Joanna Strug, Katedra Informatyki Technicznej, Wydział Inżynierii Elektrycznej i Komputerowej, Politechnika Krakowska.
Paper submitted for publication: 29.04.2010

Designation

OOS	–	Object Oriented System
ONM	–	Object Net Model
O_j	–	j^{th} object modeled within ONM
$^{i}TP_j$	–	i^{th} test path for object O_j
$^{i}TS_j$	–	i^{th} test scenario for object O_j
ev_j	–	i^{th} event of a test path
pre-$conf_j$	–	pre-configuration of ev_i
post-$conf_j$	–	post-cofiguration of ev_i
val_j	–	input or output data of ev_i
c-class	–	equivalence class for configurations of an object
d-class	–	equivalence class for input and output data of an event

1. Introduction

A widely accepted standard that assists developing object-oriented systems (OOSs) is the Unified Modeling Language. “The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components” [31]. UML makes it possible to describe various aspects of a system. These can be sized into five fundamental views: use case, process, design, implementation and deployment. The views are interrelated and cover almost whole development of a system. However, under development a system should be carefully validated and tested. Before the beginning of the implementation a system should be validated against specification requirements to check whether or not “a proper system is to be designed”. Under implementation a system should be tested to answer the question whether or not its implementation meets its specification.

A system is valid if its specification correctly expresses all business requirements of a user. Animation of the specification can be used to check its correctness [9]. This requires human assistance to evaluate results of the animation, but still it can be executed with the help of test scenarios. Being certain that the specification meets user requirements, the validation of a system at any development stage can be done with regard to that specification. To this end simulation of the system and its models may be used [19].

A new method of a semi-automatic generation of test paths supporting validation of OOSs was introduced in [25]. The method focused on objects appearing in an OOS and on their interactions. As the objects execute all operations of the system, they can be seen as subsystems with their own attributes and operations. To validate the system the subsystems and their interactions are validated. In [25] an Object Net Model (ONM) and its application for generation of test paths were presented. The test paths represent behaviors of the objects that ought to be checked during a validation. The test paths ought to be transformed into test scenarios to be directly applied for validation. The aim of this paper is to outline the transformation process.

The paper is organized as follows. In section 2 we briefly discuss related work. Section 3 states the problem. Section 4 gives an overview of the method and sections 5 describes the main steps of obtaining test scenarios from test paths. The paper ends with conclusions.

2. Previous work

In recent years many researchers have turned their attention to various UML models as a potential source of information for generating test cases supporting validation or testing of OOSs. Most of these researchers have focused on testing what is concerned only fully implemented units or implemented and integrated systems. Only a few have dealt with validation.

Generally the following three levels of OOS testing may be distinguished: unit level, integration level and system level.

At *the unit level* a behavior of individual instances of classes is tested. Usually UML state diagram is used to this end. The diagram describes states of an object and transitions between the states, including hierarchy and conditions. Thus, it shows how external events affect the behavior of the object. The method presented by Abdurazik and Offutt [1] consists in transformation of UML state diagrams into state transition tables, enumeration of transition predicates and deduction of test cases satisfying various coverage criteria. An extension to their work, which supports handling of call and signal events and four types of actions, was provided by Briand et al. [5]. Kim et al. [17] converted UML state diagrams into Extended Finite Machines (EFMs) and Flow Graphs, and then applied conventional FSM traversing algorithms and data analysis techniques. EFMs obtained from UML state diagrams were intermediate models proposed also by Ma et al. [18]. They applied branch cutting method to traverse them. Test cases developed according to the method are supposed to reveal requirement faults and functional defects in components. Sequence diagrams were also processed in order to generate tests being adequate for class testing, although not as often as state diagrams. Javed et al. [16] described a method that consists in manual development of so called SMC model (Sequence of Method Calls) from sequence diagrams, and its progressive transformations accordingly to a set of rules. The final product was a platform-specific set of test cases to be used within JUnit or SUnit testing framework. An approach that considered both UML state diagrams and sequence diagrams was proposed by Sokenou [27]. Each sequence diagram was considered to be a set of test cases, and the state diagrams attached to it provided states information for participating objects. Generated test cases may be used for class, as well as for integration testing.

Integration level testing concerns interaction between objects previously implemented and tested independently. Objects interact with each other using method calls and signal passing. Hence, UML interaction diagrams (sequence or collaboration) were the main, but often not the only one, artifacts used in approaches aiming at generation of integration tests. Abdurazik and Offutt [2] derived such tests by applying traditional control and data flow analysis to collaboration diagrams. Fraikin and Leonhardt [10] used a restricted form of sequence diagrams complemented by test data (parameters and return values for method calls) as a test specification for Java programs. In works by Pelliccione et al. [22] and Ali et al. [3] collaboration diagrams and UML state diagrams were combined to generate test cases. Pelliccione et al. [22] used collaboration diagrams to describe test directives and state diagrams to describe behavior of objects. Ali et al. [3] mapped both diagrams into a model called SCOTEM (State COllaboration TEst Model) that was traversed to derive test paths (later supplemented manually with selected test data). The results of experimental evaluation of their method showed that it ensures high fault detection for certain classes of faults (also defined in this work).

At the *system level testing* consists in comparing an implementation of a system with its specification. In published papers the methods of test generation were based on various UML diagrams. Use case diagrams were used by Salem and Balasubramaniam [24] and by Hartmann et al. [14]. Both approaches shared the idea of transforming use cases scenarios into state diagrams, activity diagrams [14] or a Flow Graph [24] which were used as base artifacts for derivation of test cases. Unfortunately, these scenarios were represented in a textual form, and their transformation required manual processing. However, use case diagrams give a general overview of a functionality of the system, and thus they are a good source of information being useful at high level testing and validation. The lack of formality can be overcome by merging the information with the ones coming from other types of UML diagrams. Briand and Labiche [6] described the TOTEM methodology for deriving functional system test requirements from use case diagrams and sequence diagrams associated with them. At first, sequential dependencies among use cases were captured in a form of activity diagrams, and then legal sequences of use cases were generated. Obtained sequences were the first component of a test plan. The next component was a collection of operation sequences identified on the basis of sequence diagrams. To define the collection various sequences of messages were considered. Finally, both components were combined together to form test requirements scenarios that could be transformed into test cases, test oracles, and test drivers if detailed design information were available. Also Sarma and Mall [26] proposed an approach that integrates use case diagrams and sequence diagrams. It consisted in transforming use case diagram into a Use case Diagram Graph (UDG) and sequence diagrams into a Sequence Diagram Graphs (SDGs). Nodes of SDGs represented messages together with their corresponding objects and they stored information regarding them, such as attributes of objects, parameters of methods and predicates. Eventually, SDGs were integrated into UDG to form a single system testing graph (STG). The STG was then traversed to enumerate paths, which constituted test cases. Generated tests are suitable for detection of use case dependency, interaction and scenario faults. However, only a restricted form of sequence diagrams is accepted. An approach of Riebisch et al. [23] was based on use cases refined by state diagrams that describe the entire system (not individual objects). State diagrams were transformed into usage models which describe system behavior and usage. Obtained usage model served as an input for test cases derivation. Information concerning operational (intended) use of a system, stored within this model, helped to determine which parts of the system may need the most thoroughness testing, and thus it helped to select an adequate set of test cases. However, this approach aims at statistical testing rather than at fault detection. Hyungchoul et al. [13] presented a method that uses only activity diagrams. An ordinary activity diagram was converted into an I/O Activity Diagram (IOAD), which exposed only external inputs and outputs of a system. From IODA a directed graph was constructed to extract test cases. To increase efficiency of system testing the test cases were derived accordingly to all-paths coverage criterion.

Validation is “the process of evaluating system or component during or at the end of the development process to determine whether a system or component satisfies specified requirements” [30]. Methods on validating class diagrams against requirements expressed in Object Constraints Language (OCL) [29] were presented by M. Clavel and M. Egea [7] or by M. Gogolla et al. [11]. Both works shared the idea of using so called snapshots and checking if they satisfy the defined invariants. An approach by F. Javed et al. [15] was also dedicated to validation of class diagrams and used snapshots, but it involved converting UML representations into an equivalent grammar and parsing manually provided scenarios. An approach dedicated to validation of design models was presented by Dinh-Trong et al. [8]. It used sequence diagrams and class diagram. Relevant information from both types of the diagrams was incorporated into a Variable Assignment Graph (VAG). Symbolic execution techniques were then applied to it to derive test input constraints which were solved by constraint solver Alloy. It results in an expected set of test cases that are suitable for detecting faults commonly introduced by designers (missing branch, faulty condition, wrong references to parent classes, etc.). Pilskalns and Andrews [21] also used information derived from sequence and class diagrams to validate UML models. Sequence diagrams were converted into Object Method Directed Acyclic Graphs (OMDAGs), and a class diagram was used to define partitions and boundaries for attributes and method parameters. Test sequences were then derived by traversing paths in OMDAGs and supplementing the paths with values selected from partitions. Detailed information referring to the generated sequences was recorded in an Object Method Execution Table (OMET). The resultant test can reveal some structural and logical errors across various UML models. However, the methods were either limited to validation of selected models only or focused more on checking consistency of different models rather than on checking these models against requirements. A formal method supporting validation of a specification against user requirements was proposed by Truong and Souquieres [28]. They transformed class and sequence diagrams (describing scenarios) into B specification and then checked it relatively to defined properties. Only safety and liveness properties were considered in their work. This alone may not be enough to validate all behavioral aspects of a system. In [12] Hesling et al. introduced a technique that uses reverse engineering to convert test specification into a use case model. Test cases re-generated from the model were shown as activity diagrams where activities represented test steps (annotated with information relating them to various test data). Such test cases are easier to review than the test specification itself. However, as they were based on a test specification instead of a requirements specification, they may not exactly reflect required functionality of a system.

3. Problem Statement

Active elements of an OOS are objects. They cooperate with each other by exchanging messages. An event occurs when an object receives or sends a message. An object reacts on a received (an observed) event by activating one from provided services and by sending (generating) an event (Fig. 1). It is assumed here that an internal component of an object (an implementation of an object) is not known.

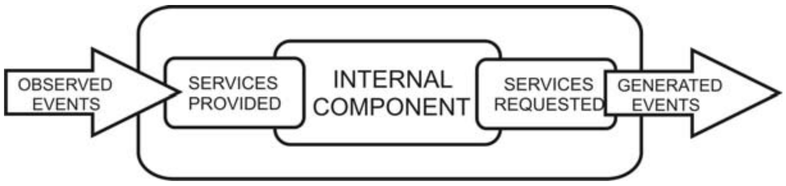

Fig. 1. Architecture of an object

Rys. 1. Architektura obiektu

Behavior of an OOS is a product of behaviors of all objects belonging to the system. Hence, the validation of the objects and their interactions is equivalent to the validation of the system. An OOS is valid if all its objects and their interactions are valid.

For validation of interacting objects their behavior in real environment should be simulated. This can be done at object level (as an Object Net Model – ONM) because there is no limit on the type and the number of exchanged messages. Contrary to hardware systems there are no extra "wires" between objects imitating actual wires connecting hardware modules that should be tested at system level. This guarantees that any event observed by the validated object will be correctly passed to it when required.

Each object within an ONM is modeled in a way that indicates how the object cooperates with other objects of OOS. Hence, the objects may be considered separately without losing the ability to validate also their interaction. Test scenarios can be used for validation of the interacting objects. A test scenario consists of a sequence of events received and sent by an object and of test data (values assigned to attributes of the object and values assigned to parameters of the events). To validate the objects one set of test scenarios for each of them is needed. Each of the sets should be *complete* and *reduced*. A set is complete when it includes test scenarios triggering each of the events observed by an object at least ones and produces each of the events generated by the object at least once. Such a set encompass all of the events necessary to determine if the object meets specification requirements. A set is reduced if the total length of all test scenarios is minimal or semi-minimal.

The aim of our research is to provide a method for semi-automatic generation of complete and reduced sets of test scenarios for all objects of a system, thus for the validation of the system against specification requirements. The method is based on UML models of a system which are worked out during elaboration of user requirements and requirements analysis that is at the very beginning of the development of the system. Use case diagrams, classes, object diagrams, and sequence diagrams are used for generation of test paths. Then, class diagrams are applied for supplementing the test paths with appropriate test data.

4. The Method Overview

The method consists of the following three stages (Fig. 2):
– development of ONM,
– generation and reduction of test paths [25],
– transformation of test paths into test scenarios (section 5).

At the very beginning of Stage 1 (Fig. 2) *classes* and *object diagrams* are analyzed to enumerate objects. C*lasses* provide one representative object for each class. O*bject diagrams*, if any, provide specific objects that differ from typical instances of the classes. All of them complete a set of objects that should be validated. Then Object Net Model (ONM) for the system is created. It consists of net models (subONMs), each representing complete behavior of one of the enumerated objects. The subONMs are developed basing on information extracted from *sequence diagrams* and *use case diagram*. *Sequence diagrams* provide sequences of events affecting each object. *Use case diagram* reveals relationships among services provided by the system to its users and hence points out connections between sequences of events related to different *use cases*. This helps to assemble the subONMs.

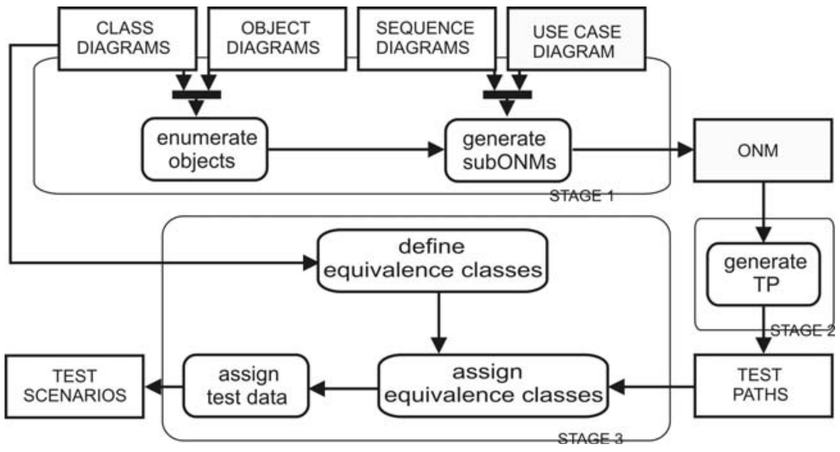

Fig. 2. Overview of the method

Rys. 2. Zarys metody

To validate an interacting object it is required that:

1) each event observed by the object will be triggered at least once,
2) each event generated by the object will be produced at least once during the simulation (or animation).

In Stage 2 (Fig. 2) ONM is analyzed to generate one set of test paths for every object of the system. A test path includes a sequence of events that shows what an object does when it participates in execution of system functions. A set of test paths for a given object is obtained in two steps. Both steps include procedures that help to reduce the size of the resultant sets of test paths. At the same time, the approach guaranties that all different events affecting the behavior of each object are represented by the test paths.

In Stage 3 (Fig. 2) each test path is transformed into a test scenario. To this end, events labeling edges of each test path are supplemented with test data. Test data consists of values assigned to attributes of an object and values assigned to parameters of events. Values of attributes determine configurations, in which an object should be to enable triggering of given events. Values of parameters determine input and output data for testing. The domain of test data is restricted by *a class diagram*. However, application of an exhaustive validation would be inadequate, because in general the number of possible test data can be infinitive. Therefore, first the domains of configurations and input and output data are divided into finite sets of equivalence classes [19]. Two configurations or two input and output data belong to one equivalence class iff both can validate the same functionality of an object. The equivalence classes should be defined by a test designer.

After partitioning test data into equivalence classes, each test path is traversed and selected equivalence classes are assigned to its events. The aim of this step is to determine what kinds of configurations and input and output data are required to enable and trigger functionalities of the object that are represented by each test path. Then, during the second traversing of test paths, appropriate test data representing the equivalence classes are selected for events of the test paths. Assignment of equivalence classes and test data is performed semi-automatically. The assignment of proper test data guaranties that each event will appear when the test scenarios will be applied. Hence, the last two stages ensure fulfillment of the requirements 1 and 2.

5. Generation of test scenarios

A test scenario ${}^{i}\mathrm{TS}_j$ for an object O_j is a sequence of quadruples:

$$(\{\text{pre-conf}_1, \mathrm{ev}_1, (val_1, val_2, ..., val_l), \text{post-conf}_1\},$$

$$\{\text{pre-conf}_2, \mathrm{ev}_2, (val_1, val_2, ..., val_k), \text{post-conf}_2\}\ldots,$$

$$\{\text{pre-conf}_i, \mathrm{ev}_i, (val_1, val_2, ..., val_l), \text{post-conf}_i\},$$

$$\{\text{pre-conf}_{i+1}, \mathrm{ev}_{i+1}, (val_1, val_2, ...), \text{post-conf}_{i+1}\},\ldots,$$

$$\{\text{pre-conf}_m, \mathrm{ev}_m, (val_1, val_2, \ldots, val_l), \text{post-conf}_m\}),$$

where
- ev_i is an event observed or generated by O_j,
- val_1, val_2, ..., val_l are input data (passed to O_j) or output data (passed by O_j) assigned to parameters of ev_i, and

pre-conf_i and post-conf_i are configurations of O_j before and after it receives or generates ev_i. For every two subsequent events, ev_i and ev_{i+1}, post-conf_i and pre-conf_{i+1} are the same because configuration of an object can be changed only when an event occurred.

Test scenarios for objects of OOS are obtained from test paths, generated previously for these objects as described in [25]. A test path ${}^{i}\mathrm{TP}_j$ generated for object O_j shows how the object behaves (in terms of events observed and generated by O_j) when a particular system functionalities are executed. Such test path is a path derived from the subONM_j which defines the behavior of O_j. It consists of edges and links. Edges of ${}^{i}\mathrm{TP}_j$ represent events observed and generated by O_j. As an event occurs when an object receives or sends a message, the description of the event refers to the description of the message. Thus, an event of the test path is described by: a name and parameters of the corresponding message, (usually also the method), a type (*call*, *ret*, etc.), a direction (*in* when the event refers to a received message, *out* when it refers to a sent message, and *out-in* for self messages), and a name of the sender or receiver (an object sending or receiving the message). A label of the edge may also include a guard condition for an event. Links of the test path reflect connections between different system services and they are labeled with pre- and post-conditions defined for the services [25]. Table 3 includes a description of test path ${}^{1}TP_3$ generated for object O_3 (an instance of class EC from the class diagram in Fig. 3). The 1st column of the table lists edges and links of the test path, the 2nd column describes events labeling the edges and the 3th column gives the guard conditions of the events or conditions assigned to links of ${}^{1}TP_3$ (compare with Fig. 5 in [25]).

The transformation of a test path into a test scenario consists of two steps:

– an assignment of equivalence classes to events of the test path,
– an assignment of test data to the events.

Equivalence classes define what kind of configurations and input and output data are needed to enable and trigger particular sequence of functionalities. Thus, the aim of the first step is to select equivalence classes that define the range of proper test data for each subsequent event of a test path representing these functionalities. The information is used in the second step to aid a selection of the actual test data.

Configurations of all objects and input and output data for all events should be partitioned into their respective equivalence classes before the transformation of any test path begins. It is due to the fact that events tackled by a given object refer to methods of the object as well as to method of other objects.

5.1. Test data partitioning

Equivalence classes are defined manually basing on a class diagram determining domains of attributes and of parameters of methods, constraints imposed on the class diagram, and on a general knowledge of the system being designed. Fig. 3 shows a part of a simple class diagram (for a simplified elevator system). It includes only classes and constraints that were used to explain how to transform test path ${}^{1}TP_3$ into test scenario ${}^{1}TS_3$.

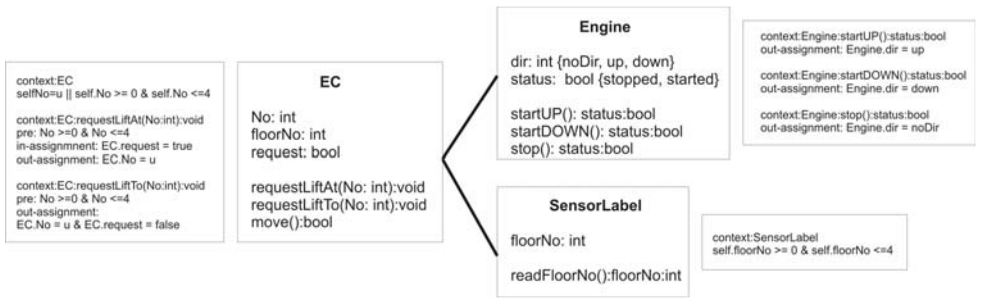
context:EC
selfNo=u || self.No >= 0 & self.No <=4
context:EC:requestLiftAt(No:int):void
pre: No >=0 & No <=4
in-assignmnent: EC.request = true
out-assignment: EC.No = u
context:EC:requestLiftTo(No:int):void
pre: No >=0 & No <=4
out-assignment:
EC.No = u & EC.request = false
EC
No: int
floorNo: int
request: bool
requestLiftAt(No: int):void
requestLiftTo(No: int):void
move():bool
Engine
dir: int {noDir, up, down}
status: bool {stopped, started}
startUP(): status:bool
startDOWN(): status:bool
stop(): status:bool
context:Engine:startUP():status:bool
out-assignment: Engine.dir = up
context:Engine:startDOWN():status:bool
out-assignment: Engine.dir = down
context:Engine:stop():status:bool
out-assignment: Engine.dir = noDir
SensorLabel
floorNo: int
readFloorNo():floorNo:int
context:SensorLabel
self.floorNo >= 0 & self.floorNo <=4

Fig. 3. Class diagram and its constraints

Rys. 3. Diagram klas i nałożone na niego ograniczenia

The constraints should include:

– invariants related to attributes and associations,
– *pre-* and *post-conditions* of methods,
– *in-assignment* and *out-assignment* defined for methods.

It is here assumed that an attribute and a parameter having the same name will pass their values to each other when a method is invoked or terminated (a default assignment). Otherwise it is necessary to define explicit how invoking and terminating methods change values of attributes of an object and values of parameters of the methods. The *in-* and *out-assignment* clauses are used to this aim. For example, when the method *requestLiftAt(No:int):void* (Fig. 3) is invoked it should change values of two attributes of O_3: *No* and *request*. The value of parameter *No*, that is passed by the method to O_3 will be assigned to the attribute *No* (a default assignmemts). The expected change for attribute *request* is defined within the in-assignment clause related to the method.

Test data consists of two components: configurations of objects and input and output data of events. Therefore, two types of equivalence classes are introduced:

– *c*-classes – equivalence classes for configurations of objects,
– *d*-classes – equivalence classes for input and/or output data of events.

Configurations of an object O_j are determined by values of its attributes. The configurations are partitioned into a set of disjunctive c-classes: {*c*-class$_1$, …, *c*-class$_l$, …, *c*-class$_n$}, such that *c*-class$_l$ = $\{(va_1, ..., va_m) \mid va_i \in \text{sub-dom}_k(a_i)\ \&\ va_1\sim...\sim va_m\}$, where va_i denotes a symbolic value of attribute a_i, ~ denotes relation between these values and sub-dom$_k(a_i)$ is a subdomian of a_i.

Table 1 shows the domains and sub-domains defined for attributes of O_3 and Table 2 lists the *c*-classes identified for the object.

Table 1

Domains and sub-domains for attributes of O_3 (instance of EC)

a_i	dom(a_i)	sub-dom$_1$(a_i), …, sub-dom$_k$(a_i)
No	$\{u\} \cup \{0, 1, 2, 3, 4\}$	$\{u\}$, $\{0, 1, 2, 3, 4\}$
floorNo	{0, 1, 2, 3, 4}	{0, 1, 2, 3, 4}
request	{false, true}	{false}, {true}

Table 2

Equivalence classes for configurations of O_3 (instance of EC)

c-class$_i$	definition of c-class$_i$
c-class$_1$	{(*vNo*, *vfloorNo*, *vrequest*) | *vNo* ∈ sub-dom$_1$(No) & *vfloorNo* ∈ sub-dom$_1$(floorNo) & *vrequest* ∈ sub-dom$_1$(request)}
c-class$_2$	{(*vNo*, *vfloorNo*, *vrequest*) | *vNo* ∈ sub-dom$_1$(No) & *vfloorNo* ∈ sub-dom$_1$(floorNo) & *vrequest* ∈ sub-dom$_2$(request)}
c-class$_3$	{(*vNo*, *vfloorNo*, *vrequest*) | *vNo* ∈ sub-dom$_2$(No) & *vfloorNo* ∈ sub-dom$_1$(floorNo) & *vNo* = *vfloorNo* & *vrequest* ∈ sub-dom$_2$(request)}
c-class$_4$	{(*vNo*, *vfloorNo*, *vrequest*) | *vNo* ∈ sub-dom$_2$(No) & *vfloorNo* ∈ sub-dom$_1$(floorNo) & *vNo* < *vfloorNo* & vrequest ∈ sub-dom$_2$(request)}
c-class$_5$	{(*vNo*, *vfloorNo*, *vrequest*) | *vNo* ∈ sub-dom$_2$(No) & *vfloorNo* ∈ sub-dom$_1$(floorNo) & *vNo* > *vfloorNo* & *vrequest* ∈ sub-dom$_2$(request)}

Each event ev_i is related to some method m_i, because it occurs when the method is either invoked or terminated. Hence, the input and output data of ev_i are also the input and output data of the related method m_i and are determined by values of parameters of the method. The input and output data of ev_i (m_i) are partitioned into one set of disjunctive *d*-classes {*d*-class$_1$, …, *d*-class$_l$, …, *d*-class$_m$}. A *d*-class is defined in the same manner as a *c*-class, but the definition refers to domains and values of parameters of a method.

5.2. Assignment of equivalence classes

Each test path $^i\mathrm{TP}_j$ describes some behavior of O_j by showing events involved in execution of functionalities related to the behavior. The equivalence classes are selected and assigned to the events to determine what kind of test data will guarantee their triggering.

To adhere to the definition of a test path, three equivalence classes are assigned to each event ev_i labeling an edge of $^i\mathrm{TP}_j$:

– *c*-class$_k$/pre – a *c*-class that will provide pre-conf$_i$ for ev_i,
– *c*-class$_j$/post – a *c*-class that will provide post-conf$_i$ for ev_i, and
– *d*-class/ – a *d*-class that will provide input or output data for ev_i

The equivalence classes extend labels of nodes and edges of $^i\mathrm{TP}_j$, as shown in Figure 4.

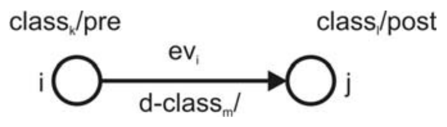

Fig. 4. Assignment of equivalency classes to ev_i labeling an edge (i, j)

Rys. 4. Sposób przypisania klas równoważności do zdarzenia ev_i krawędzi (i, j)

The equivalence classes are selected and assigned to events of a test path during a traversing of the test path. The final result of the assignment is one equivalence class of each kind assigned to each event of the test path, but during the execution of the selection procedure subsets of equivalence classes may be temporary assigned to the events.

The selection of c-class/pre, d-class/ and c-class/post for event ev_i is aided by analyzing the following information:

– guard condition of ev_i (if any),
– condition assigned to a link, if the edge labeled by ev_i follows or precedes the link,
– pre-conditions or post-conditions of a method related to ev_i,
– default assignment and in-assignments or out-assignments of the related method,
– preceding equivalence class.

Different combinations of the above information are used when a particular equivalence class is selected. It depends on the type (call, return, etc.) and direction (in, out, out-in) of the considered event and on the kind of the currently selected equivalence class (*c*-class/pre, *c*-class/post or *d*-class). For example, *c*-class/pre assigned to event *in(FButton)call:requestLiftAt(No)* should satisfy the conditions labeling link (NULL, 3.1.0), the d-class should also satisfy the pre-conditions of *requestLiftAt(No:int):void*, and the c-classes/post for the event was selected basing on in-assignment and default assignment of the related method.

C-classes/pre selected for an event ev$_i$ and *c*-classes/post selected for the preceding event ev$_{i-1}$ should be the same (it results from the definition of a test scenario). To ensure fulfillment of the requirements the subset of c-classes/post of ev$_{i-1}$ (the preceding equivalence class) is compared with the subset of c-classes/pre selected for ev$_i$ and if they are different their intersection is assigned to both events. Moreover, when the former equivalence class was restricted, it is also possible to start a recursive refinement of earlier assignments. For example, the comparison made after selecting the *c*-class/pre for event *out(Door)call:open()* makes it possible to removed two *c*-classes (the crossed ones) from the subset of *c*-classes/post assigned previously to the event *in (ELight)ret: switchON:ON)* and start the recursive refinement that lets also the subset of *c*-classes/post assigned to *requestLiftAt(No:int):void* be further restricted.

Table 3 summarizes the assignments done for ^{1}TP$_3$. Some verses of the table were removed as their content brings nothing of importance to the way equivalence classes are selected. The first three columns of the Table describe the test path. Additionally, in the 3th column, relevant constraints defined for methods related to the events are given. The last two columns show the results. Most of the *c*-classes were selected automatically. The crossed ones were first selected and then removed as the result of the recursive refinement. Only the underlined *c*-classes were selected manually. Selection of *d*-classes for all events is here unambiguous and in most cases the *d*-classes do not contribute when selecting *c*-classes. Hence, they are not shown in Table 3.

Table 3

Assignment of equivalence classes for events of ^{1}TP$_3$

(i, j)	ev$_i$	conditions and constraints	/pre	/post
(NULL, 3.1.0)	–	EC.No = *u* & EC.request = false		
(3.1.0, 3.1.1)	in (FButton)call: requestLiftAt(No)	pre: No≥0 & No≤4 in-assign: EC.request=true	*c*-class$_1$	*c*-class$_3$, ~~*c*-class$_4$~~, ~~*c*-class$_5$~~
(3.1.1, 3.1.2)	out (ELight)call: switchON()		*c*-class$_3$, ~~*c*-class$_4$~~, ~~*c*-class$_5$~~	*c*-class$_3$, ~~*c*-class$_4$~~, ~~*c*-class$_5$~~
(3.1.2, 3.1.3)	in (ELight)ret: switchON:ON		*c*-class$_3$, ~~*c*-class$_4$~~, ~~*c*-class$_5$~~	*c*-class$_3$, ~~*c*-class$_4$~~, ~~*c*-class$_5$~~
(3.1.3, 3.1.4)	out (Door)call: open()	EC.No=EC.floorNo	*c*-class$_3$	*c*-class$_3$
(3.1.4, 3.1.5)	in (Door)ret: open:opened		*c*-class$_3$	*c*-class$_3$
(3.1.5, 3.1.6)	out (FButton)ret: requestLiftAt:	out-assign: EC.No=*u*	*c*-class$_3$	*c*-class$_2$
(3.1.6, 3.2.0)	-	EC.No = *u* & EC.request = true		
(3.2.0, 3.2.1)	in (EButton)call: requestLiftTo(No)	pre: No≥0 & No≤4	*c*-class$_2$	~~*c*-class$_3$~~, ~~*c*-class$_4$~~, *c*-class$_5$

continuous tab. 3

(*i*, *j*)	ev$_i$	conditions and constraints	/pre	/post
(3.2.1, 3.2.2)	out (Door)call: close()		~~*c*-class$_3$~~, ~~*c*-class$_4$~~, *c*-class$_5$	~~*c*-class$_3$~~, ~~*c*-class$_4$~~, *c*-class$_5$
(3.2.2, 3.2.3)	in (Door)ret: close:closed		*c*-class$_5$	*c*-class$_5$
(3.2.3, 3.3.0)	-	EC.No≠EC.floorNo		
(3.3.0, 3.3.1)	out-in (EC)call: move()		~~*c*-class$_4$~~, *c*-class$_5$	~~*c*-class$_4$~~, *c*-class$_5$
(3.3.1, 3.3.2)	out (Engine)call: startup()	EC.No>EC.FloorNo	*c*-class$_5$	*c*-class$_5$
(3.3.2, 3.3.3)	in (Engine)ret: start:started		*c*-class$_5$	*c*-class$_5$
(3.3.3, 3.3.5)	out (SensorLabel)call: readFloor()		*c*-class$_5$	*c*-class$_5$
(3.3.5, 3.3.6)	in (SensorLabel)ret: readFloor:floorNo		*c*-class$_5$	~~*c*-class$_3$~~, *c*-class$_4$, *c*-class$_5$
(3.3.6, 3.3.7)	out (SensorLabel)call: readFloor()	EC.No≠EC.floorNo	*c*-class$_4$, *c*-class$_5$	c-class$_4$, *c*-class$_5$
(3.3.7, 3.3.8)	in (SensorLabel)ret: readFloor:floorNo		*c*-class$_4$, *c*-class$_5$	~~*c*-class$_3$~~, ~~*c*-class$_4$~~, *c*-class$_3$
(3.3.8, 3.3.9)	out (Engine)call: stop()	!(EC.No≠EC.floorNo)	*c*-class$_3$	*c*-class$_3$
(3.3.9, 3.3.10)	in (Engine)ret: stop:stoped		*c*-class$_3$	*c*-class$_3$
(3.3.10, 3.3.11)	out–in (EC)ret: move:true		*c*-class$_3$	c-class$_3$
(3.3.11, 3.2.3')	–	EC.No=EC.floorNo		
(3.2.3', 3.2.4)	out (Door)call: open()		*c*-class$_3$	*c*-class$_3$
(3.2.9, 3.2.10)	out (EButton)ret: requestLiftTo:	out-assign: EC.No = *u* & EC.request = false	*c*-class$_3$	*c*-class$_1$
		EC.No = *u* & EC.request = false		

The information supporting selection that is incorporated into the selection procedure makes it possible to limit the number of possible equivalence classes for each event. However, the procedure may failed to narrow their number up to one *c*-class/pre, one *d*-class and one *c*-class/post for each event. Thus, it may be necessary to make the final refinement manually (as shown for events *in(SensorLabel)ret:readFloor:floorNo* and *out(SensorLabel)call:readFloor()* from Table 3).

5.3. Selection of test data

To complete the transformation of test paths into test scenarios the events of the test paths have to be supplemented with appropriate test data. Test data are real values assigned to attributes of objects and parameters of events. The values, selected for each event, should be representative for their corresponding equivalence classes and ensure executability of the generated test scenario.

The procedure adding test data to events of test paths is very similar to the one used for assigning equivalence classes. It also requires the transformed test paths be traversed, and also uses information provided by class diagram and test paths (default and explicit assignments, different conditions, preceding test data). The procedure additionally uses the equivalence classes assigned in the previous step to limit the number of possible test data for each event.

Table 4 gives test scenario ${}^{1}TS_3$ for object O_3. It shows events, configurations and input and output data that have been selected for events of ${}^{1}TP_3$.

Table 4

Test scenario ${}^{1}TS_3$ for object O_j

ev_i	pre-$conf_i$	input/output data	post-$conf_i$
in (FButton)call: requestLiftAt(No)	(*u*,0,false)	(0)	(0,0,true)
out (ELight)call: switchON()	(0,0,true)	–	(0,0,true)
in (ELight)ret: switchON:status:ON	(0,0,true)	(ON)	(0,0,true)
out (Door)call: open()	(0,0,true)	-	(0,0,true)
In (Door)ret: open:status:opened	(0,0,true)	(opened)	(0,0,true)
out (FButton)ret: requestLiftAt:	(0,0,true)	-	(*u*,0,true)
in (EButton)call: requestLiftTo(No)	(*u*,0,true)	(2)	(2,0,true)
…			
Out (SensorLabel)call: readFloor()	(2,0,true)	–	(2,0,true)
in (SensorLabel)ret: readFloor:floorNo	(2,0,true)	(1)	(2,1,true)
Out (SensorLabel)call: readFloor()	(2,1,true)	–	(2,1,true)
in (SensorLabel)ret: readFloor:floorNo	(2,1,true)	(2)	(2,2,true)
out (Engine)call: stop()	(2,2,true)	–	(2,2,true)
in (Engine)ret: stop:status:stoped	(2,2,true)	(stoped)	(2,2,true)
Out–in (EC)ret: move:-:true	(2,2,true)	(true)	(2,2,true)
out (Door)call: open()	(2,2,true)	–	(2,2,true)
…			
out (EButton)ret: requestLiftTo:	(2,2,true)	–	(*u*,2,false)
– in *out-in (EC)ret: move:–:true* denotes lack of named returned parameter			

The test data for each event of a test path being transformed into a test scenario are first restricted by the equivalence classes assigned to the events. Then the other information is taken into consideration. They are used basically in the same way as they were during assignment of the equivalence classes. For example, default and in-assignment were checked when post-conf$_1$ (0,0,true) for event *in(FButton)call:requestLiftAt(No)* was selected.

In most cases configurations and output data can be selected automatically. For example, the post-conf of the event *in(SensorLabel)ret: readFloor:floorNo* passing value 1 is set to (2,1,true), because the value should be assigned to the attribute *floorNo* (basing on default assignment for *floorNo*) and values of the remaining two attributes should not change (basing on the lack of out-assignment for *readFloor()* and on preceding configuration). Input data usually need to be inserted by a test designer, unless it is uniquely determined by its equivalence class or explicitly given by an event. For example, the event *in (Engine)ret:stop:status:stoped* provides explicitly the value *stopped* for its parameter *status*, but a value passes to the object by *in (SensorLabel)ret: readFloor:floorNo* must be given by a test designer. Hence, also the selection of test data is a semi-automatic process.

6. Conclusions

For validation it is essential to provide test scenarios that may be applied early at the development of a system and be used systematically during the whole development process. To achive this goal it is necessary to use high level description of the system, such as UML models. The concept of using UML is not new. Our method shares with [6] or [26] the idea of merging information extracted from use case diagram and sequence diagrams to get a view of the whole behavior of a system. However, in contrast to the methods, our approach focuses on individual objects executing different system functionalites as well as on interaction of the objects. It gives us the opportunity to generate separate sets of test scenarios for each of the objects, but without losing the possibility to validate both: behavior of the objects and their cooperation.

Most steps of the method are automated. However, certain assistance of a test designer is needed, especially during the transformation of test paths into test scenarios. The involvement of a test designer could be here reduced, when detailed information about how objects process data were provided.

An important advantage of the method is an application of systematic procedures aiming at reducing the size of generated sets of test scenarios. So far this problem was mention by only a few researchers [1, 6, 20] but their approaches were dedicated mostly to testing not to validation of a system. Moreover, the reductions were limited to rejecting redundant tests after finishing their generation [1, 20] or to stopping random generation of tests when assumed coverage for some criterion was achieved [6]. In our method various reduction procedures are applied throughout the whole generation process. The majority of the reductions are performed during generation of test paths (see [25]). However, the application of equivalence partitioning and selection of test data representing the equivalence classes causes that only one test scenario for each test path is generated. Such test scenario is enough to validate the behavior of the system related with the test path.

The method aims at providing means to validate all behaviors of a system, without repetition. In our future work we are going to implement the method and get some experimental results regarding its accuracy and efficiency.

References

[1] Abdurazik A., Ofutt J., *Generating Tests from UML Specifications*, 2nd International Conference on Unified Modeling Language, Fort Collins, USA 1999, 416-429.

[2] Abdurazik A., Ofutt J., *Using UML Collaboration diagrams for static checking and test generation*, 3th International Conference on Unified Modeling Language, York, UK 2000, 383-395.

[3] Ali S., Briand L.C., Jaffar-ur-Rehman M., Asghar H., Zohaib M., Iqbal Z., Nadeem A., *A State-based Approach to Integration Testing based on UML Models*, J. Information & Software Technology, 49 (11-12), 2007, 1087--1106.

[4] Binder R.V., *Testing Object-Oriented Systems – Models, Patterns, and Tools*, Addison-Wesley, 1999.

[5] Briand L.C., Cui J., Labiche Y., *Toward Automated Support for Deriving Test Data from UML statecharts*, 6th International Conference Unified Modeling Language, San Francisco, USA, 2003, LNCS, Vol. 2863, Springer, Heidelberg, 2003, 249-264.

[6] Briand L.C., Labiche Y., *A UML-based Approach to System Testing*, J. Software and System Modeling 1(1), 2002, 10-42.

[7] Clavel M., Egea M., *ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Static Class Diagrams*, 11th International Conference on Algebraic Methodology and Software Technology, Kuressaare, Estonie 2006, 368-37.

[8] Dinh-Trong T.T., Ghosh S., France R.B., *A Systematic Approach to Generate Inputs to Test UML Design Models*, 17th International Symposium on Software Reliability Engineering, Raleigh, USA 2006, 95-104.

[9] Dockerill K., *The Importance of Animation with UML*, 9th International Symposium of the INCOSE, Brighton, UK 1999.

[10] Fraikin F., Leonhardt T., *SeDiTeC – Testing Based on Sequence diagrams. 17th IEEE International Conference on Automated Software Engineering*, Edinburgh, UK 2002, 261-266.

[11] Gogolla M., Bohling J., Richters M., *Validating UML and OCL Models in USE by Automatic Snapshots Generation*, 6th International Conference Unified Modeling Language, San Francisco, USA 2003, LNCS, Vol. 2863, Springer, Heidelberg 2003, 265-279.

[12] Hasling B., Goetz H., Beetz K., *Model Based Testing of System Requirements using UML Use Case Models*, 1st International Conference on Software Testing, Verification, and Validation, Lillehammer, Norway, 2008, 367-376.

[13] Hyungchoul K., Sungwon K., Jongmoon B., Inyoung K., *Test Cases Generation from UML Activity Diagrams*, 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, Quingdao, China 2007, 556-561.

[14] Hartmann J., Vieira M., Foster H., Ruder A., *A UML-based Approach to System Testing*, Journal of Innovations System Software Engineering, Vol. 1, 2005, 12-24.

[15] J a v e d F., M a r n i k M., B r y a n t B.R., G r a y J., *A Grammar-Based Approach to Class Diagram Validation*, 4th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools, St. Louis, USA, 2005, 45-51.
[16] J a v e d A.Z., S t r o o p e r P.A., W a t s o n G.N., *Automated Generation of Test Cases Using Model-Driven Architecture*, 2nd International Workshop on Automation of Software Test, Minneapolis, USA 2007, 3.
[17] K i m Y., H o n g H., B a e D., C h a S., *Test Cases Generation from UML State Diagrams*, IEEE Proceedings – Software 146 (4), 1999, 187-192.
[18] M a Y., L u J., Z h a o R., *A Test Path Generation Approach for Component Testing based on UML State Diagram*, International Conference on Software Engineering and Applications. Dallas, USA 2006.
[19] M y e r s G.J., S a n d l e r C., B a d g e t t T., T h o m a s T.M., *The art of testing*, Wiley, 2004.
[20] N g P., *A Concept Lattice Approach for Requirements Validation with UML State Machine Model*, 5th International Conference on Software Engineering Research, Management & Applications, Busan, South Korea 2007, 393-400.
[21] P i l s k a l n s O., A n d r e w s A., F r a n c e S., G h o s h R., *Rigorous Testing by Merging Structural and Behavioral UML Representations*, 6th International Conference on the Unified Modeling Language, San Francisco, USA 2003, 234-248.
[22] P e l l i c c i o n e P., M u c c i n i H., B u c c h i a r o n e A., F a c c h i n i F., *TeStor: Deriving Test Sequences from Model-based Specifications*, 8th International SIGSOFT Symposium on Component-based Software Engineering, St. Louis, USA 2005, LNCS Vol. 3489, Springer, Heidelberg, 2005, 267-282.
[23] R i e b i s c h M., P h i l i p p o w I., G o t z e M., *UML-Based Statistical Test Case Generation*, International Conference on Objects, Components, Architectures, Services, and Applications for Network World, LNCS 2591, 2003, 394-411.
[24] S a l e m A.M., B a l a s u b r a m a n i a m L., *Utilizing UML use cases for testing requirements*, International Conference on Software Engineering research and practice, Las Vegas, USA 2004.
[25] S a p i e c h a K., S t r u g J., *Automatic Test Paths Generation from UML Models*, 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques, Kraków, Poland 2009, 115-128.
[26] S a r m a M., M a l l R., *Automatic Test Case Generation form UML Models*, 10th International Conference on Information Technology, Roulkela, India 2007, 196-201.
[27] S o k e n o u D., *Generating Test Sequences from UML Sequence Diagrams and State Diagrams*, 2nd Workshop on Model-Based Testing, Vienna, Austria 2006, 236-240.
[28] T r u o n g N.-T., S o u q u i e r e s J., *Validation of UML scenarios using B prover*, 3th Taiwanese-French Conference on Information Technology, Nancy, France 2006.
[29] W a r n e r J., K l e p p e A., *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.
[30] IEEE Std. 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, IEEE, 1990.
[31] Unified Modeling Language Specification, version 1.4.2, 2005 (http://www.omg.org/).