Computing and Informatics, Vol. 40, 2021, 772–795, doi: 10.31577/cai_2021_4_772

ACTIVITY DIAGRAM GENERATION BASED ON USE-CASE TEXTUAL SPECIFICATION

Bogumiła Hnatkowska, Mateusz Cebinka

Faculty of Information and Communication Technology
Wrocław University of Science and Technology
Wyb. Wyspiańskiego 27
50-370 Wrocław, Poland
e-mail: bogumila.hnatkowska@pwr.edu.pl

Abstract. The requirements specification phase is one of the most important during software development. In many cases, its outcome takes a form of a use-case model, which consists of use-case diagrams and supplementary use-case specifications. The requirements specification document is used by various stakeholders, starting from customers or their representatives, through architects, developers to testers. Each role may have specific preferences for the form of requirements specification. To solve this problem, we propose a template for writing use-cases based on the existing guidelines and a transformation method that creates an activity diagram from the use-case textual specification consistent with the proposed template. There are several tools that can generate activity diagrams based on textual specification, but none of them fully meets the requirements for the form of template or resulting diagram, which should be correct (textual specification semantics preserved), UML 2.5 syntax compliant and contain necessary data. The proposed transformation method is supported by a tool that transforms models at the same level of abstraction. The transformation itself is defined at the meta-model level. The general idea of model-to-model transformation is not new, but the meta-models are original and fit for purpose. The application of the method is demonstrated by several examples. Due to the frequent potential changes in created specifications, the automation of the process will save time. Moreover, a graphical representation of a use-case is easier to analyze and find errors or inconsistencies compared to a textual specification.

Keywords: Requirements specification, textual specification, use-case template, activity diagram, generation

1 INTRODUCTION

Requirements specification plays a key role in software development. This phase and its outcomes are always present, regardless of the methodology used. In many cases, the specification document takes a form of a use-case model that consists of use-case diagrams and supplementary use-case definitions. The quality (readability, consistency, completeness) of this specification is crucial to the project's success. Quality issues are addressed, among others, through the use of templates when a textual specification is created or through a graphical representation of the specification, e.g., by an activity diagram, for which certain quality checks can be performed automatically.

The requirements specification document is used by various stakeholders, starting from customers or their representatives, through architects, developers ending with testers. Each role may have specific preferences for the appearance of the requirements specification. Customers are likely to prefer a textual form over formal diagrams because of its comprehensibility and ease of creation. Developers or testers would probably prefer diagrams over textual specifications due to their unambiguity and the potential application of generation tools.

To solve the problem of different needs for the form of use-case specification, we propose a textual template for writing use-cases based on the existing guidelines and a transformation method that creates an activity diagram from the textual use-case specification assuming it follows the proposed template.

There are several methods and tools that perform similar tasks, but none of them fully meets our requirements, both in terms of the supported textual specification template and the generated activity diagram, which should be informative, consistent with UML 2.5 syntax [20] and correct (the semantics of the textual specification should be preserved).

We also expect the solution to be runnable in a popular commercial modeling tool available to students because of its educational value. Therefore, it was implemented as a plug-in to Visual Paradigm as a multi-stage transformation of models at the same level of abstraction. The general idea of model-to-model transformation is not new, but the meta-models are original and fit for purpose.

The rest of the paper is structured as follows. Section 2 discusses existing guidelines for writing 'effective use-cases' and approaches for translating textual specifications into activity diagrams. Section 3 presents the proposed solution and Section 4 the way of its evaluation. The last Section 5 concludes the work.

2 RELATED WORKS

Since the concept of use-cases emerged, many authors have tried to define effective ways of their specification, e.g., [1, 3, 14]. Recommendations include, among others, some general suggestions like 'use a simple grammar'. Some propose more specific solutions in the form of use-case templates, e.g., [1, 14]. These books were the

inspiration for defining assumptions about the use-case specification considered in the transformation. The details are presented in the next section.

In [13], the authors propose a semi-automatic approach to generate use-case scenarios based on parametrized use-case patterns. The proper use-case pattern should be selected by the system analyst depending on the use-case goal. The generated use-case specification contains the main flow and extensions. The steps are uniquely identified. Alternative flows are denoted by letters, e.g., 'a'. Steps within a particular alternative flow are numbered separately. Another way of step's identification has been proposed in [18]. Actions from the main flows are rewritten in alternative scenarios instead of being reused. The substituted actions in alternative flows are identified with prim, which is not very readable. On the other side, the RSL proposed in [18] allows definition of pre- and post condition for use-cases. Our approach to step identification will use a method similar to [13]. As confirmed in the experiment described in [24], the application of step identifiers to describe variation points are best understood by different stakeholders (better results in comparison to tags, specific sections, or advice use-cases). As it is done in the RSL [18], we are going to allow defining pre- and post-condition for use-cases and invocation of another use-cases.

Transformation of a textual use-case specification to an activity diagram is the subject of many papers. Some of them assume that the textual specification does not use any specific template and is written in free natural language, e.g., [10, 12]. The authors follow typical steps for NLP processing such as text tokenization, parts-of-speech tagging, stemming and lemmatization, type dependencies, and information extraction to perform the transformation. The first solution [10] is defined for English, the second [12] for Arabic. They look for specific sentence patterns, e.g., SVO (subject verb object), for which transformation rules are applied. The transformation rules are defined in natural language. Unfortunately, in both cases no example or implemented tool is presented.

The NLP approach has been used in [16], but here the use-case is assumed to follow a specific template defined in a table, including pre- and post-conditions, main flow, alternative flows, and unique identifiers for the steps. The transformation rules generating a sequence diagram are defined as a mixture of natural language and pseudo-code sentences. The proposed method will use a very similar template for use-case specification. NLP techniques do not produce satisfactory results in terms of diagram correctness and completeness.

A use-case specification template, like that described in [16], is also used in [5]. However, some of the well-defined rules for use-case specification are not applied, e.g., sometimes the subject of action is unknown, 'if' statements are allowed. The input (use-case specification) and output (activity diagram) of the method are formally defined – the formal definitions form a kind of meta-model. Transformation rules are defined informally, in natural language, based on patterns found in the text. The transformation helps to detect possible errors in the use-case specification in the early modeling stages. Identified errors include the reference to a non-existent step identifier or the absence of an alternative flow. The transformation method

proposed in this paper has the same features.

The activity diagram shown in [5] does not contain partitions. It consists of activity nodes labeled with a step number and a verb (action) optionally followed by processed objects (given in parentheses). In our approach we will use partitions, but information about the data flow is omitted (it can be partially retrieved from the action name).

A slightly different template is proposed in [11]. An example of a textual use-case specification is defined at the subfunction level. It focuses on exceptions, neglecting alternative flows. Transformation rules are defined in pseudo-code, but the authors do not mention their implementation even in a prototype tool.

In the paper [23], which is a systematic review of transformation approaches between user requirements and analysis models, the authors identify constraints on requirements specification satisfaction of which is necessary to keep the transformation correct. They include, among others, the following recommendations:

- use simple grammar,
- use active rather than passive voice,
- use the same verb for the same action,
- do not use pronouns,
- use specific constructs to model the control flow of events (GOTO STEP [number], CON [statement], IF-THEN, WHILE-ENDWHILE, REPEAT [number] UNTIL, DO-UNTIL).

The RUCM approach to use-case specification [22] also defines a set of keywords for specifying control structures in a scenario:

- conditional (IF THEN ELSE),
- concurrent (MEANWHILE),
- condition checking (VALIDATES THAT),
- iteration (DO-UNTIL),
- stop (ABORT or RESUME).

Another set of control structures is defined in [18]. There are conditions ($-\rangle$ cond), invocations ($-\rangle$ invoke), and final statements (success/failure). The RSL allows a single use-case to have many multiple flows whose actions, if necessary, should be linked by a special statement ('rejoin').

The proposed approach will also use a set of keywords to model the control flow. The list of control structures is limited (we have excluded IF THEN ELSE, as this structure is not recommended in e.g. [3], and MEANWHILE – no concurrent activities are supported at this time). The keywords are slightly different.

Unfortunately, the activity diagram generated in the RUCM method does not have partitions and has ill-defined conditions (on the decision nodes instead on the branches itself). The transformation rules are defined in natural language (in tables) referring to meta-models defined internally in the aToucan tool.

The papers [9, 8] are further examples of studies in which the activity diagram is either generated at a very general level (in [9] it consists of only one activity) or violates syntax rules (in [8] it has ill-defined guards and looks rather like a flow-chart than an activity diagram).

A meta-model representing use-cases is also defined in [17]. As the authors claim, such a meta-model can have many different representations, including activity diagrams. The authors have defined in a table how various elements of the meta-model can be presented (transformed) by the elements of activity diagram. In this solution, neither decision nor merge nodes are used to represent control flows. Instead, the authors propose to use guards directly on control flows. There are no partitions on the activity diagrams. The authors also do not consider iterations.

An extended version of the meta-model from [17] is presented in [19]. This meta-model is used to represent parametrized patterns of use-cases. The logic of these patterns is demonstrated with activity diagrams. The diagrams include so called 'insertion points' where other pattern instances can be included. We decided that it is too early to consider parametrized use-case specification in the transformation process.

Three tools that generate an activity diagram from a textual use-case textual were found and thoroughly checked. That were: Visual Paradigm (VP) [21], CaseComplete [2] and Enterprise Architect (EA) [4]. All limit their interest to the control flow. Action pins can be generated by EA, but they must be explicitly defined for steps. None of them can generate swimlanes.

VP assumes that the scenario is defined in pseudocode (with *if then, while, jump* statements), which breaks the recommendation for writing efficient use-cases. The generated activity diagram rather resembles a flowchart with conditions defined nearby the branch symbol – compare Figure 1.

CaseComplete supports most of the good practices defined for textual use-case specification [1, 3, 14]. The only problem found is the inability to define nested alternative flows (alternatives for alternatives). The steps in main flow as well in the extensions are identified by numbers, however the numbers in the alternative flows are not unique (they always start with 1). It is possible to define many alternative scenarios for a single step. The loop definition requires a specific statement, i.e. 'continues', followed by the step number from the main flow. Similarly, the fact that a scenario (sub-scenario) ends should be directly noted by the verb 'end'.

The activity diagram generated by CaseComplete tool looks much better than that the one generated by Visual Paradigm. The tool does not process the text of the specification in any way – actions in the activity diagram contain whole sentences from the scenario along with the step number. Loops in the activity diagram are presented indirectly as links to steps. To model many alternative flows starting in the same branch, CaseComplete uses a fork symbol with many outgoing arrows with guards (see Figure 2). Unfortunately, the guards are always taken from the trigger of the alternative scenario, which can lead to semantically incorrect activity diagrams (see Figure 3). According to the diagram, the user first selects a card, and then the condition that user selects bank transfer is checked.

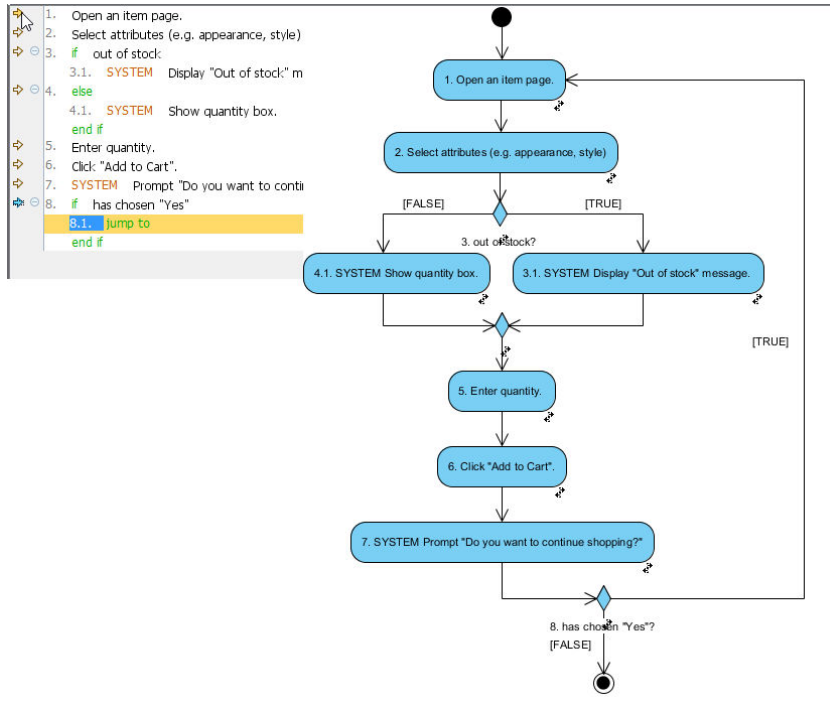

Figure 1. Visual Paradigm – an example of activity diagram generation [7]

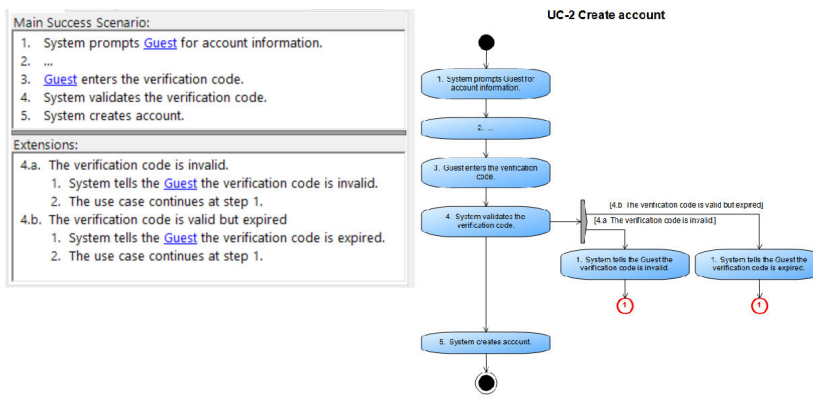

Figure 2. CaseComplete – an example of activity diagram generation with a loop and many alternative flows

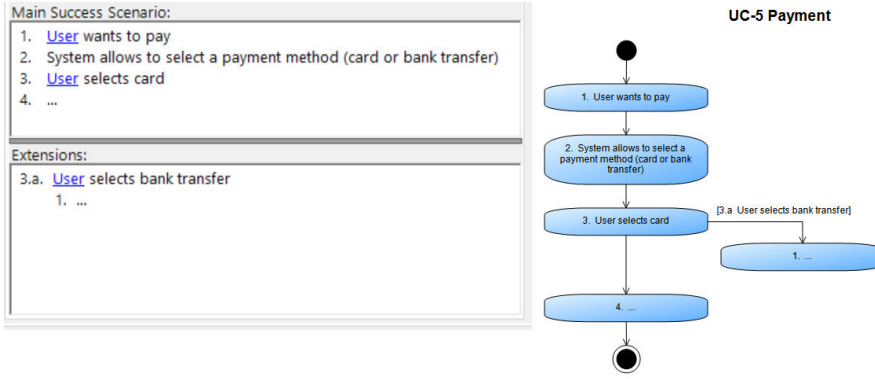

Figure 3. CaseComplete – an example of an improperly generated activity diagram

Enterprise Architect generates an activity diagram from a structured use-case specification. The structured editor is a little bit cumbersome, especially when a use-case has multiple scenarios. The tool allows to define many alternatives starting in the same step. The name of the alternative is taken as a guard. As in CaseComplete, it is impossible to define an alternative to an alternative scenario. It is impossible to define loops within the specific scenario level (main, alternative). Loops are not generated when the alternative scenario contains no action beyond information where to join the main scenario. In consequence, the activity diagram presented in Figure 11 cannot be generated. The generated diagrams do not have merge nodes, which results in semantics errors (the activity 'System asks about the payment method' cannot be started without tokens in all incomming branches) – see Figure 4 as an example.

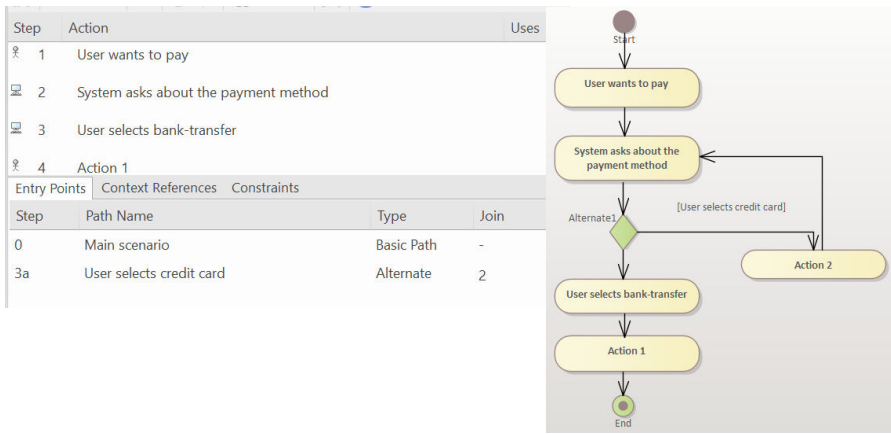

Figure 4. Enterprise Architect – an example of an improperly generated activity diagram

3 PROPOSED SOLUTION

3.1 Assumptions

The purpose of the research is to propose a transformation from a textual use-case specification to its presentation as an activity diagram. This subsection collects the main assumptions about the input and output of the transformation method.

Textual use-case specification should be consistent with the template defined in the next subsection. It should allow definition of the main flow of events along with alternative flows in structured natural language (English) (Scenario Plus Fragments [1]). It should be possible to define alternative flows to alternative flows (one nesting level), and loops in any scenario levels.

Sentences that are part of the use-case specification should follow the recommendations found in the literature, among others:

1. Use-cases are written from the user perspective (the user goal level).
2. Actions are defined using the VSO pattern [3] in the active voice – it is clear who or what performs the action [3, 14].
3. Steps have unique identifiers.
4. Every alternative scenario requires a detectable condition to be defined [1].
5. Use-case inclusions or extensions are written as casual actions.
6. Specification is technology neutral [1, 14].

The original semantics of the use-case specification should be preserved during the transformation. The activity diagram should describe an event flow, not a data flow. Its syntax must conform to the UML 2.5 specification [20]. Branches are the only control structure considered. Actors should be organized into swimlanes.

3.2 Use-Case Textual Template

This section describes a proposed template for textual use-case specification.

The template consists of three tables describing:

1. General information about the use-case.
2. Main flow of events.
3. Alternative flows of events – it there are no alternative flows, the table is empty.

The type of each table must be defined in its first row (Main Use Case, Main Flow, Alternative Flow).

The first table (*T1*) additionally contains the name of the use-case and optionally – the use-case pre- and post-conditions (see Table 1).

The second table (*T2*) describes the main flow of events (main scenario). The table is divided into many columns (one per acting entity, i.e., actor or system).

Main Use Case	
Name	Borrow book
Precondition	Librarian is logged in
Postcondition	Book is marked as borrowed

Table 1. General use-case description

Activities are organized within numbered rows (their identifiers are given in the first column). They can be subdivided into numbered steps (actions). If there is only one action in a row, its number can be omitted. It is assumed that the numbers in this table start at 1 and are increased by 1 in each row (see Table 2). However, the numbers serve for identification purpose only. The ordering of actions is inferred from their positions in the template – preceding actions are executed before their successors. Actions can only be assigned to one agent (actor or system) in one row.

Main Flow		
	Librarian	System
1	Librarian wants to register a book borrowing to a specific reader.	
2		System asks for reader ID and book ID.
3	Librarian enters reader ID and book ID.	
4		1. System verifies the reader ID exists. 2. System verifies that the book ID exists.
5		System assigns the book ID to the reader ID.

Table 2. Main flow of events – an example

Actions are represented by statements in English written according to the SVO template. The statements should be written in the simple present tense. The only exception is the first action in an alternative flow, which should be written in the simple past tense. It is treated as a trigger for the alternative scenario.

Five types of statements are supported, from which repetitions, conditional, and goto statements must contain specific phrases described by regular expressions (the solution is case-insensitive):

- Conditional statements in the present tense: {verifies | validates}[that | the].
- Conditional statements in the past tense: {verified | validated}[that | the].
- Repetitions: {repeats until}.
- Goto statement: {goto step}.
- Casual statements.

Casual statements can contain any verb followed by the name of an active entity (actor or system) except those mentioned above, e.g., 'verifies', 'repeats'. In particular, it can be the verb 'run' or 'call' followed by the name of another use-case. This construction can be used to denote relationships between use-cases.

The goto, conditional, and repetition statements are linked with activities/steps of the same or other scenarios. Each goto statement must be followed by the identifier of an existing step.

Alternative scenarios, if any, must be described in a table identified by the term 'Alternative flow' – see Table 3. The rows in this table represent activities. The first column identifies the activity of the main scenario to be affected. The step identifiers are numbered according to the pattern $X.Y$ (first level alternative scenarios) or $X.Y.Z$ (second level alternative scenarios – alternative scenarios to alternative scenarios), where:

- X – a number being a reference to an action identifier from the specific activity of the main flow, e.g., 2.
- Y – a letter identifying a thread within the scenario, e.g., a.
- Z – a letter describing a step identifier in the sub-sub scenario, e.g., a, b, c.

The entire alternative scenario should be placed in a single column assigned to a specific actor; however, actions may refer to other actors listed somewhere (as column names).

Alternative Flow		
	Librarian	System
4		1. a. System verified that the reader ID does not exist. 2. a. System informs that the reader ID does not exist. 3. a. System verifies that the reader ID is a number. 4. a. Goto step 2.
4		2. b. System verified that the book ID does not exist. 3. b. System informs that the book ID does not exist. 4. b. Goto step 2.
4		3. a. a. System verified the reader ID is not a number. 3. a. b. System informs the reader ID is not a number. 3. a. c. Goto step 2.

Table 3. Alternative use-cases description – an example

Three alternative scenarios are defined in Table 3. All refer to activity 4 from the main flow (Table 2). The first alternative scenario replaces steps 2–4 with their new versions 2. a.–4. a. Then, processing moves to activity 2 from the main flow. Similarly, the second alternative scenario introduces a new thread 'b' consisting of steps 4–6. The last row introduces an alternative flow to the first alternative flow (containing step 4. a). After two steps 'a' and 'b', the flow returns to activity 2 of the main flow.

An alternative scenario requires a detectable condition to be defined elsewhere. At that moment, only binary conditions are supported. Such a condition must be defined after the verb 'verifies' or 'validates'. This makes the specification slightly longer, but it is still flexible and readable. For example, it is not impossible to define an extension in the way suggested in [3], because there is no condition defined here in the main flow:

4. User has the system saved the work so far
...
4. a. Save fails
...

Instead, the specification should look like this:

4. User has the system saved the work so far
5. System verifies that the save was successful
...
5. a. System verified that the save has failed
...

The definition of a loop requires the presence of both a conditional statement and an alternative scenario with a 'goto' action. It is not very convenient, but such specification was supported by the meta-model defined. An example of such a construction is given in Table 4 and Table 5.

Main Flow		
	Actor	System
1	Action 1	
2.		Action 2
3.	1. Repeats until condition	
4.		Action 3

Table 4. Main flow with repetition statement

Alternative Flow		
	Actor	System
3	1. a. Goto step 1.	

Table 5. Alternative flow for repetition statement

3.3 Transformation Process

The transformation is a multi-stage process. It goes through three meta-models (see Figure 5):

1. *M1* – Use-case template meta-model,
2. *M2* – Activity diagram meta-model,
3. *M3* – Visual Paradigm (VP) meta-model.

The textual use-case specification is read into an instance of the *M1* meta-model. The *M1* meta-model is an object-oriented representation of tables and their rows. Its instance is translated to an instance of the activity diagram meta-model (*M2*) and then translated to an instance of the meta-model *M3*. The last meta-model is used to facilitate the implementation of the activity diagram visualization in the Visual Paradigm tool. Meta-models and transformation rules are defined in the subsections below.

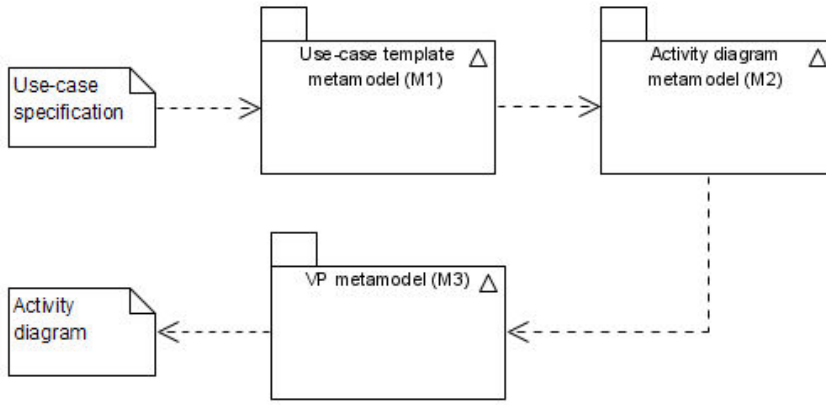

Figure 5. Transformation workflow

3.4 Meta-Models

M1 is the first meta-model in the chain of model transformations (see Figure 6). This model is an object-oriented representation of the use-case textual template. The *UseCase* class contains the main data about the use-case, i.e. its name, pre- and post-conditions if they are presented. It contains a collection of *Row* instances each of which has a unique identifier. The row represents a singular step. The *Identifier* has the following attributes:

- *mainId* – activity identifier, e.g., 1, within the main flow of events;
- *stepId* – action identifier, e.g., 1, within the activity; can be inferred;
- *altStepId* – an optional letter, e.g., 'a', describing a plot within the alternative flow;

- *identifierId* – an optional number or letter (for sub-scenarios) describing a step identifier within a specific plot, e.g., 1 or 'b'.

Each row contains data about the actor performing the action, the text describing the action (*originalText*) and its shorter version (without numbers), and the type of scenario to which the row belongs to (main, alternative or alternative to alternative).

If the action contains a goto statement, an instance of the *GoTo* class is created. It stores the identifier to the target row.

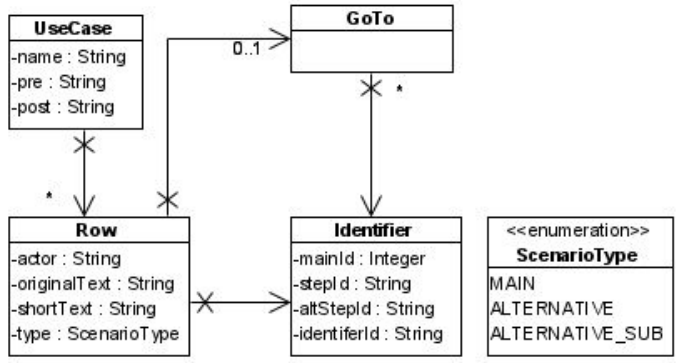

Figure 6. *M1* – Use-case template meta-model

For the flows from Tables 6 and 7, the instance of the *M1* meta-model will look as shown in Figure 7. As can be seen, no knowledge inference is performed during parsing tables with text. As the result, two identical identifiers are created (*i1* and *i4*), even though they intentionally represent the same entity.

Main Flow		
	Actor	System
1		1. System verifies the reader ID exists

Table 6. Main flow with conditional statement

M2 is a meta-model that serves as a representation of a simplified description of activity diagrams – see Figure 8. An activity diagram is a graph consisting of nodes. The abstract class *Node* is the root for all considered node types: *ActionNode*, *MergeNode*, and *DecisionNode*.

The nodes are described by:

- *text* – name of the action,
- *actor* – name of the actor performing the action,

Alternative		
Flow		
	Actor	System
1		1. a. System verified that the reader ID does not exist. 2. a. Goto step 1. 1.

Table 7. Alternative flow with conditional statement

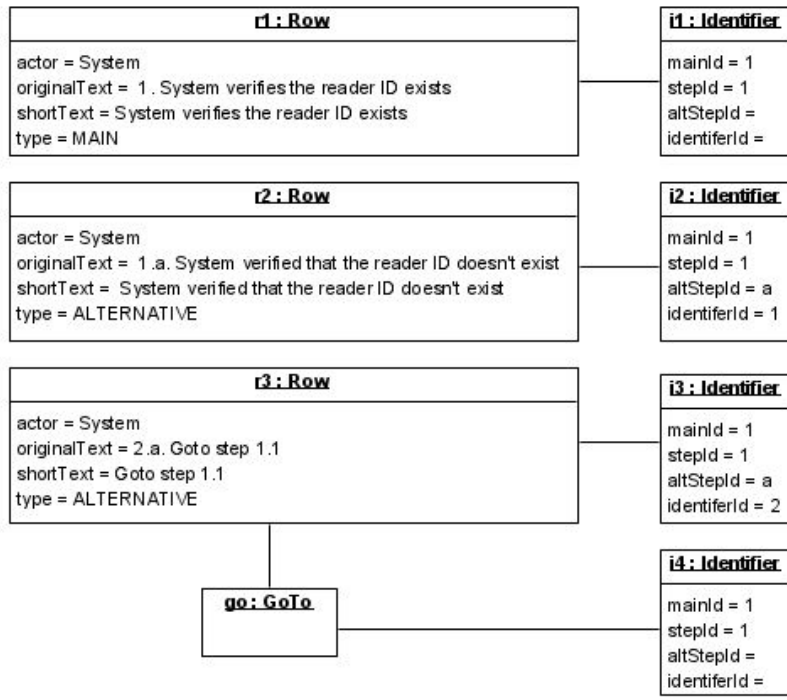

Figure 7. An example of *M1* meta-model instance

- *isEnd* – Boolean equal to *true* if the node is a final node.

The meanings of *Identifier* and *ScenarioType* are the same as in *M1*.

Concrete nodes keep references to their successors. The *DecisionNode* class stores up to two such references for two possible branches the guards of whose are kept in the *Edge* class instances. This class additionally has the attribute *isLoop* set to *true* when its instance starts a loop.

The *M3* is the target meta-model for displaying activity diagrams in the Visual Paradigm. The *M3* meta-model is very similar to the *M2* meta-model – compare Figure 9. The abstract *Node* class has been replaced by the abstract *Element* class, whose attributes refer to the tool API.

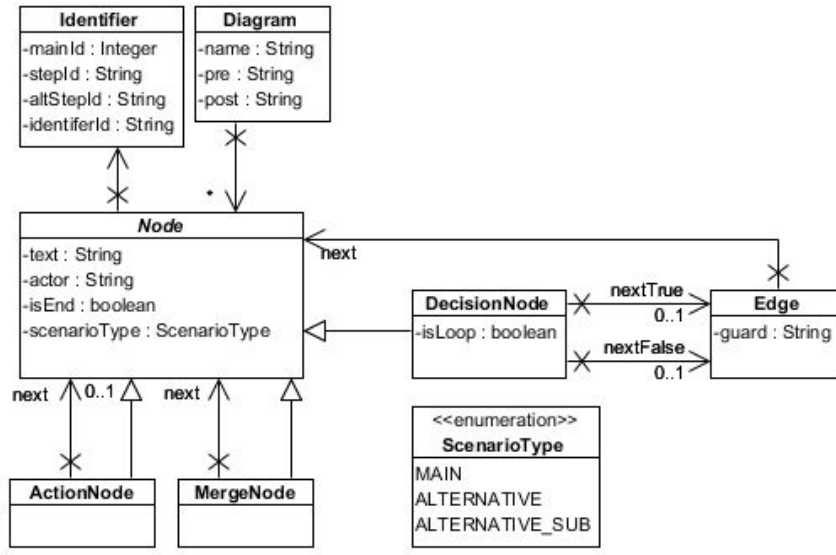

Figure 8. *M2* – Activity diagram meta-model

The list of concrete classes inherited from the *Element* class has been extended to include *FinalNodeElement* and *InitialNodeElement* classes representing final and initial nodes, respectively. The same happened to the list of strings representing guards stored by a *DecisionNodeElement* instance – now the strings are stored in one place for both outgoing branches.

The *Diagram* meta-class has remained the same. It now stores a reference to a swimlane object which is split into one or more partitions. The *Partition* instance groups together the elements (actions) performed by a specific actor.

3.5 Transformation Rules

This section gathers transformation rules between meta-model instances. The rules have unique identifiers that determine their order. Those with lower identifiers are run first. Rules are defined in natural language mixed with pseudo-code. Some of them will be illustrated with examples.

3.5.1 M1 to M2 Transformation Rules

Rule 1 – Map UseCase to Diagram. Create an instance of *Diagram* class for the instance of *UseCase* class and copy the attributes.

Rule 2 – Map Rows to Nodes. Each *r: Row* with repetition or conditional sentence in the present tense in *shortText* is mapped to a *a* instance of the *Deci-*

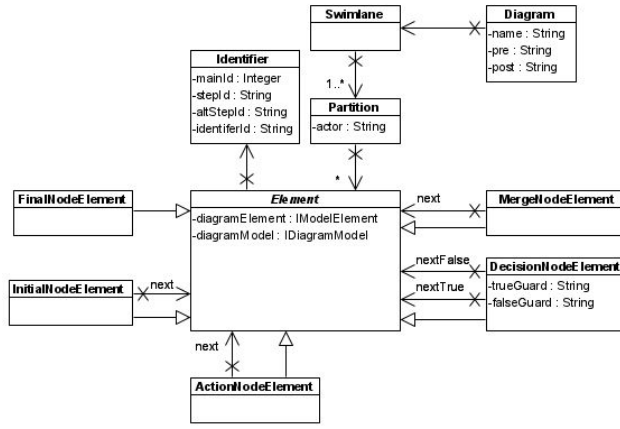

Figure 9. *M3* – Visual Paradigm meta-model

sionNode class; rows with other types of sentences are translated to instances of the *ActionNode* class.

Attribute values are copied: $a.actor = r.actor$, $a.text = r.shortText$, $a.scenarioType = r.scenarioType$. If r contains a sentence with repetition, the attribute *isLoop* is set to *true*.

Rule 3 – Create successors nodes in positive scenarios. Create an instance of the *ActionNode* class with an empty identifier and set it as the predecessor for the first node. This action will mimic the initial state.

For each *goto: GoTo* find:

- row identifier (*sourceId*) to which *goto* belongs,
- node *source* with the *sourceId* identifier,
- node *target* with the identifier that *goto* points to.

Set $source.next = target$.

For each *a: ActionNode* set the *next* attribute to the corresponding action created from the succeeding source row (if any).

For each *d: DecisionNode* create an instance e of the *Edge* class with the guard set to a new string taken from $d.shortenText$. The part before and containing 'verifies' is cut off. For example, for the sentence 'System verifies the book ID exists' the guard will be set to 'The Book ID exists'. Next, $d.nextTrue = e$ and the $e.next$ attribute is set as it is described above for $a.next$.

Rule 4 – Create successors nodes in negative scenarios. For each *n: Node* containing a conditional statement in the past tense, get the identifier and extract the *targetId* it points to. Find the decision node d with $targetId$.

Create an instance e of the class *Edge* with the guard set to 'Else'. Then, $d.nextFalse = e$ and the $e.next = n$.

Rule 5 – Create merge nodes. For each node n that is a successor to more than one another node, create an instance m of the *MergeNode* class. Create an empty identifier except of *altStepId* which is generated. All n attributes are copied to m. The successors' *next* or *nextTrue* attribute is set to m, and $m.next = n$.

Rule 6 – Create actions based on decision nodes. For each *d: DecisionNode* not being a loop create an instance a of the *ActionNode*. Create an empty *Identifier* object and link it with a Copy all the attributes from d to a (including identifier); set $a.next = d$.

Rule 7 – Delete unnecessary actions. For each *a: ActionNode* with a conditional sentence in the past tense or a goto statement, find its predecessor p. Set $p.next = a.next$ if p is an action. If p is a decision node, set the appropriate $p.nextTrue.next = a.next$ or $p.nextFalse.next = a.next$. Remove a.

Rule 8 – Find final nodes. For each *n: Node* we set $isEnd = true$ if $n.next$ is empty (for actions) or $n.nextTrue$ or $n.nextFalse$ is empty (for decisions).

An example of the transformation rules applied for the $M1$ instance shown in Figure 7 is presented in Figure 10.

3.5.2 M2 to M3 Transformation Rules

The *M3* meta-model is slightly different from the $M2$ meta-model. Therefore, the list of transformation rules is quite short. We limit the presentation to those which are not connected with the API of Visual Paradigm and for which the transformation is not trivial.

Rule 9 – Map DecisionNodes to DecisionNodeElements. For each *d: DecisionNode* create a new instance *de* of the *DecisionNodeElement* class. Set $de.trueGuard = d.nextTrue.guard$ if $d.nextTrue$ exists, $de.falseGuard = d.nextFalse.guard$ if $d.nextFalse$ exists.

Rule 10 – Create initial/final node. Find a node n with an empty identifier. Create an instance i of the *InitialNodeElement* class. Set $i.next = n.next$.

Create a list of nodes l with the attribute *isEnd* set to *false*. If the list contains any object, create a final node f. For each n in the list set its *next* field to f (if n is an action node), $nextTrue = f$ (if n is a decision node and $n.nextTrue$ is empty), and $nextFalse = f$ (if n is a decision node and $n.nextFalse$ is empty).

Rule 11 – Create *Swimlane* and Partitions. Create a new instance of the *Swimlane* class and assign it to the diagram instance. Group the nodes for each actor. Create a partition for each actor. Assign instances of the *Element* class to the partition belonging to the actor the source node points to. Initial state, decision nodes, merge nodes are assigned to the same partition to which its preceding element belongs. The final node is assigned to the first partition that contains a state leading to the final state (non-deterministic choice).

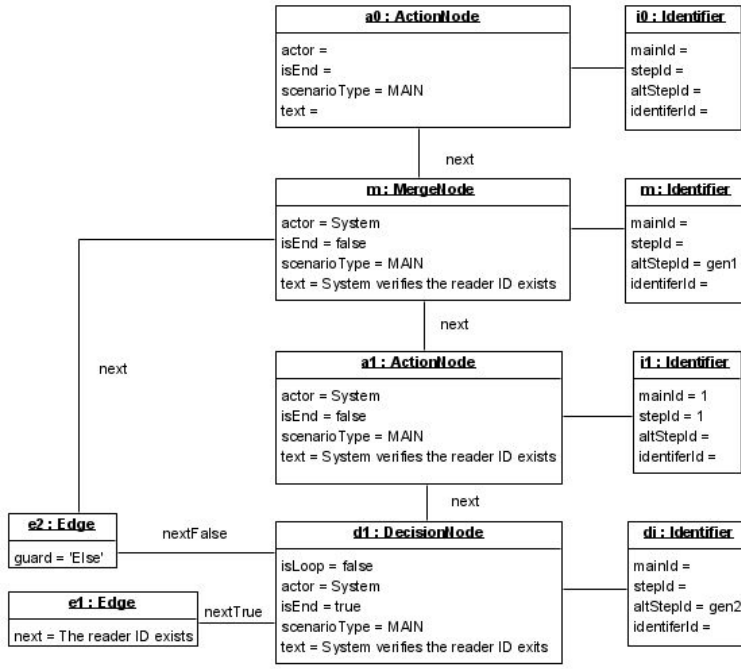

Figure 10. An exemplary $M1$ to $M2$ instance transformation

4 SOLUTION EVALUATION

4.1 Test Cases

The method was initially tested by eight test cases (four positive and four negative). The positive test cases were adapted from the external resources, mainly from [1, 3]. The use cases were chosen to cover all elements that can be generated, starting from a simple flow (case 1), through alternative flows (case 2), to loops (case 3), and multiple actors (case 4).

Negative test cases were created by injecting errors (e.g. reference to non-existent action) to positive cases to check the tool's robustness to inconsistent specifications.

4.2 Validation Setup

The solution was implemented as a plug-in to the Visual Paradigm (VP, v. 16.2). The plug-in is available at the link [6].

One of the co-authors defined a use-case diagram with textual use-case specifications in a VP project. He also run the generation. The generation results (for positive test cases) were examined by the second co-author. The elements checked

included:

- Consistency of the textual use-case specification with its original version.
- Correctness of the generated diagram at the syntax and semantics level.

For negative test cases, two elements were checked:

- Location of error injection in the textual specification.
- The error message displayed by the tool.

The tool was able to generate correct diagrams for all positive test cases. An example activity diagram for the use-case specification defined in Tables 1, 2 and 3, is given in Figure 12. Another activity diagram created based on Tables 6 and 7 is given in Figure 11. The VP project with exemplary use-case scenarios and generated diagrams can also be downloaded [6].

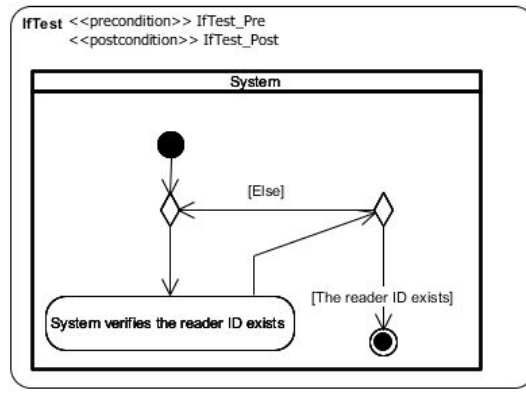

Figure 11. Activity diagram generated from the specification given in Tables 6 and 7

The tool is able to correctly identify the following possible mistakes:

- No use-case name.
- No actor name in the second row.
- No step identifier in the first column.
- Reference to non-existence action identifier.
- Starting an alternative flow with a reference to a sentence not containing the verb 'verifies'.

The tool generates the diagrams correctly, but they may not look readable. The automatic diagram layout does not work well enough.

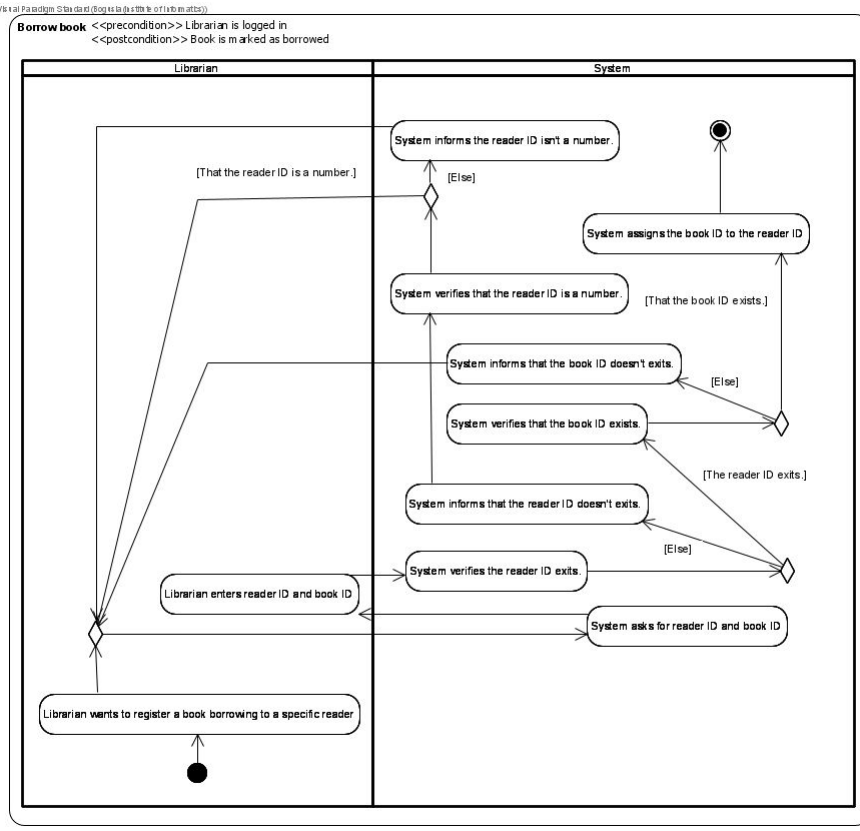

Figure 12. Activity diagram generated from the specification given in Tables 1, 2 and 3

5 CONCLUSIONS

This paper presents an approach to activity diagram generation based on a textual use-case specification. This approach requires the use-case specification to follow the template defined in Section 3. The template was proposed based on the literature overview and is adapted to the general editor available in Visual Paradigm.

The generated diagram is syntactically correct and complies with UML 2.5 syntax. It also contains necessary data.

The proposed transformation process refers three meta-models. The second meta-model is the most important because its structure constraints the transformation. An example of such a constraint is that a decision node can only have two outgoing branches at that moment. This may be inconvenient, but all scenarios can be specified.

The transformation method is supported by an implemented plug-in for Visual

Paradigm. The plug-in is available for free, and the community version of VP can be freely used for non-commercial projects.

Due to the frequent changes in the textual specifications created, automating the process will save the time of activity diagram creation. The tool is also able to identify basic errors in scenarios. This feature can be used for checking the textual specification correctness and completeness (against formal errors).

The advantages of the implemented plug-in affecting the readability and flexibility of the solution compared to other tools are:

- Use-case textual template:
  - The scenarios must be written according to the recommended rules – compare to [21].
  - All scenarios are collected in one place (page) – compare to [2, 4].
  - It is possible to define alternative flows for alternative flows – compare to [2, 4].
  - It is possible to define goto statements in any level of scenarios – compare to [2, 4].
- Activity diagram:
  - Diagrams are UML 2.5 compilant (syntax level) – compare to [2].
  - Semantics of the textual specification is always preserved – compare to [4, 21].
  - Actions are grouped into partitions – compare to [2, 4, 21].
  - Loops are modeled with decision/merge nodes – compare to [2, 4, 21].
  - Loops are visible directly. No links to steps are used to model them – compare to [2].

In the future, it is planned to extend the scenarios with the ability to define inline loops, e.g., the actor repeats steps 3–5 until the condition is met, and to define more than one alternative scenario for one step. As it comes to activity diagrams, the addition of data flow is also under consideration (at least on request) and the extension of the keyword list with the ability to express the fact that a specific branch has been completed.

REFERENCES

[1] Adolph, S.—Bramble, P.—Cockburn, A.—Pols, A.: Patterns for Effective Use-Cases. Addison-Wesley Professional, 2003.

[2] CaseComplete. Available at: `https://casecomplete.com/`.

[3] Cockburn, A.: Writing Effective Use-Cases. Addison-Wesley Professional, 2000.

[4] Enterprise Architect (Version 15.2). Available at: `https://www.sparxsystems.com/`.

[5] Liu, S.—Sun, J.—Liu, Y.—Zhang, Y.—Wadhwa, B.—Dong, J. S.—Wang, X.: Automatic Early Defects Detection in Use Case Documents. Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14), 2014, pp. 785–790, doi: 10.1145/2642937.2642969.

[6] Hnatkowska, B.: UseCase Specification to Activity Diagram Generator. Available at: `https://github.com/bhnatkowska/UseCaseToActividyDiagramTransformation`.

[7] How to Generate Activity Diagram from User-Story. Available at: `https://www.visual-paradigm.com/tutorials/user-story-to-activity-diagram.jsp`.

[8] Iqbal, U.—Bajwa, I. S.: Generating UML Activity Diagram from SBVR Rules. 2016 Sixth International Conference on Innovative Computing Technology (INTECH), IEEE, Dublin, Ireland, 2016, pp. 216–219, doi: 10.1109/intech.2016.7845094.

[9] Kamarudin, N. J.—Sani, N. F. M.—Atan, R.: Automated Transformation Approach from User Requirement to Behavior Design. Journal of Theoretical and Applied Information Technology, Vol. 81, 2015, No. 1, pp. 73–83.

[10] Maatuk, A. M.—Abdelnabi, E. A.: Generating UML Use Case and Activity Diagrams Using NLP Techniques and Heuristics Rules. International Conference on Data Science, E-Learning and Information Systems 2021 (DATA '21), ACM, 2021, pp. 271–277, doi: 10.1145/3460620.3460768.

[11] Mustafiz, S.—Kienzle, J.—Vangheluwe, H.: Model Transformation of Dependability-Focused Requirements Models. 2009 ICSE Workshop on Modeling in Software Engineering, 2009, pp. 50–55, doi: 10.1109/mise.2009.5069897.

[12] Nassar, I. N.—Khamayseh, F. T.: Constructing Activity Diagrams from Arabic User Requirements Using Natural Language Processing Tool. 2015 6th International Conference on Information and Communication Systems (ICICS), IEEE, 2015, pp. 50–54, doi: 10.1109/iacs.2015.7103200.

[13] Ochodek, M.—Koronowski, K.—Matysiak, A.—Miklosik, P.—Kopczyńska, S.: Sketching Use-Case Scenarios Based on Use-Case Goals and Patterns. In: Madeyski, L., Śmiałek, M., Hnatkowska, B., Huzar, Z. (Eds): Software Engineering: Challenges and Solutions. Springer, Cham, Advances in Intelligent Systems and Computing, Vol. 504, 2017, pp. 17–30, doi: 10.1007/978-3-319-43606-7_2.

[14] Övergaard, G.—Palmkvist, K.: Use-Cases: Patterns and Blueprints. Addison-Wesley Professional, 2005.

[15] Reggio, G.—Leotta, M.—Ricca, F.—Clerissi, D.: DUSM: A Method for Requirements Specification and Refinement Based on Disciplined Use Cases and Screen Mockups. Journal of Computer Science and Technology, Vol. 33, 2018, No. 5, pp. 918–939, doi: 10.1007/s11390-018-1866-8.

[16] Thakur, J. S.—Gupta, A.: Automatic Generation of Sequence Diagram from Use Case Specification. Proceedings of the 7th India Software Engineering Conference (ISEC '14), ACM, 2014, Art. No. 20, pp. 1–6, doi: 10.1145/2590748.2590768.

[17] Śmiałek, M.—Bojarski, J.—Nowakowski, W.—Ambroziewicz, A.—Straszak, T.: Complementary Use Case Scenario Representations Based on Domain Vocabularies. In: Engels, G., Opdyke, B., Schmidt, D. C., Weil, F. (Eds.):

Model Driven Engineering Languages and Systems (MODELS 2007). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4735, 2007, pp. 544–558, doi: 10.1007/978-3-540-75209-7_37.

[18] Śmiałek, M.—Nowakowski, W.: Presenting the Requirements Specification Language. Chapter 2. In: Śmiałek, M., Nowakowski, W.: From Requirements to Java in a Snap: Model-Driven Requirements Engineering in Practice. Springer, Cham, 2015, pp. 31–65, doi: 10.1007/978-3-319-12838-2_2.

[19] Śmiałek, M.—Ambroziewicz, A.—Parol, R.: Pattern Library for Use-Case-Based Application Logic Reuse. In: Lupeikiene, A., Vasilecas, O., Dzemyda, G. (Eds): Databases and Information Systems (DB & IS 2018). Springer, Cham, Communications in Computer and Information Science, Vol. 838, 2018, pp. 90–105, doi: 10.1007/978-3-319-97571-9_9.

[20] OMG Unified Modeling Language (OMG UML), Version 2.5, 2015. Available at: https://www.omg.org/spec/UML/2.5/PDF.

[21] Visual Paradigm. Available at: https://www.visual-paradigm.com/.

[22] Yue, T.—Briand, L. C.—Labiche, Y.: An Automated Approach to Transform Use Cases into Activity Diagrams. In: Kühne, T., Selic, B., Gervais, M. P., Terrier, F. (Eds.): Modelling Foundations and Applications (ECMFA 2010). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 6138, 2010, pp. 337–353, doi: 10.1007/978-3-642-13595-8_26.

[23] Yue, T.—Briand, L. C.—Labiche, Y.: A Systematic Review of Transformation Approaches Between User Requirements and Analysis Models. Requirements Engineering, Vol. 16, 2011, pp. 75–99, doi: 10.1007/s00766-010-0111-y.

[24] Santos, I. S.—Andrade, R. M. C.—Santos Neto, P. A.: Templates for Textual Use Cases of Software Product Lines: Results from a Systematic Mapping Study and a Controlled Experiment. Journal of Software Engineering Research and Development, Vol. 3, 2015, Art. No. 5, 29 pp., doi: 10.1186/s40411-015-0020-3.

Bogumiła Hnatkowska is Professor Assistant in software engineering at the Wrocław University of Science and Technology. She received her M.Sc. degree and her Ph.D. degree in computer science in 1992 and 1997, respectively. Her main scientific interests include but are not limited to software development processes, modeling languages, model driven development, model transformations, and quality of the software products. She is member of program committees of several international conferences. She has over 100 publications in international journals and conference proceedings from different areas of software engineering.

Mateusz Cebinka received his Bachelor degree and his M.Sc. degree in computer science in 2019 and 2020, respectively, both from the Wrocław University of Science and Technology, Poland. He has worked for three years as Java developer in the financial sector and currently at a company building IoT and biotechnology software.