Constructive-synthesizing modeling and the Process Mining methods in a toolkit to monitor and analyze the software debugging process were applied. Methods for monitoring the development and debugging processes are the basis for improving the level of practical training of students, reducing the time that is used irrationally in the process of software development by a student, and in monitoring the processes of performance of tasks by a teacher. The process of software debugging is seen as a sequence of actions when dealing with relevant tools. Using the methodology of constructive-synthesizing modeling, a constructor for forming a debugging actions log was developed. Based on the constructive model, the extension to the integrated development environment (IDE) Microsoft Visual Studio, in which all debugging actions are recorded in an event log, was designed. During debugging in the IDE, event logs are collected and then a conformance checking of these logs with regard to the reference model is performed. To do this, the ProM (Eindhoven Technical University, Netherlands), a platform for Process Mining methods, is used. By checking compliance, it is possible to compare different debugging processes and recognize behavioral similarities and differences. The main purpose of the developed toolkit is to collect debugging actions from the developer's IDE. By better understanding how students grasp and deal with errors, one can help novices learn to program. Knowing how programmers debug can encourage researchers to develop more practically directed methods, enable teachers to improve their debugging curricula and allow tool developers to adapt the debugger to the actual needs of users. It is practically suggested to use the prepared tools in the software engineering course

Keywords: Process Mining, debugging, constructive-synthesizing modeling, training, software engineering

UDC 004.9
DOI: 10.15587/1729-4061.2020.215090

DEVELOPMENT OF A TOOLKIT FOR ANALYZING SOFTWARE DEBUGGING PROCESSES USING THE CONSTRUCTIVE APPROACH

V. Shynkarenko
Doctor of Technical Sciences, Professor*
E-mail: shinkarenko_vi@ua.fm
O. Zhevaho
Postgraduate Student*
E-mail: marakonec@gmail.com
*Department of Computer and Information Technologies
Dnipro National University of Railway Transport named after Academician V. Lazaryan
Lazaryana str., 2, Dnipro, Ukraine, 49010

Received date 06.09.2020
Accepted date 22.10.2020
Published date 30.10.2020

1. Introduction

Debugging is one of the most important, complex, and time-consuming tasks in software development. The IEEE Standard Glossary of Software Engineering Terminology defines debugging as an operation to detect, localize, and correct bugs in computer programs [1]. Developers typically spend at least 30 % of their time debugging and use an integrated development environment (IDE), such as Microsoft Visual Studio [2].

Improvement of the quality of coding and debugging by students and beginning developers is the foundation for training in programming. Papers [3, 4] presented the toolkit for automatic monitoring and visualization of the code production process. These approaches are also proposed to be extended to debugging processes.

Development of effective debugging skills is especially important for beginners: they are still learning the syntax and semantics of the programming language are more likely to create a wrong code, have limited skills of understanding programs and effective debugging, which often leads to difficulties in comprehending and resolving errors [5, 6].

Empirical research into software development is most often based on the data extracted from version control systems and bug tracking tools, but not from the IDE because they do not record developers' actions. Process Mining methods make it possible to analyze process-oriented data, including the automatic discovering process models and checking the compliance of event data conform to a reference model [7].

Programming requires many competencies, and their training is a central problem in computer science education. In this regard, research aimed at forming new approaches and developing tools to improve the quality of students' training in programming is relevant. Students must not only understand programming concepts but also be able to find independent solutions when faced with bugs. Checking programs for bugs, finding, and correcting them are the main competencies of professional developers.

2. Literature review and problem statement

First, let us consider approaches to studying debugging processes.

Successful debugging usually depends on a proper understanding of the syntax and semantics of a program. The level of understanding of a program and previous experience with such bugs are two key aspects that distinguish a beginner from an experienced debugger [8, 9]. Debugging can greatly improve understanding of a program, as it requires that developers could read and understand a code. In paper [10], developers were interviewed to understand how they explore programs. The result showed that programmers spend most of their time reading and understanding a source code. The most effective tool for understanding a code, in their opinion, is multiple running of an application using a debugger. This supports the hypothesis that debugging is used not only to localize bugs but also to understand a source code.

A lecture with a learning support system was developed in article [11] to help students debug using exercises. The drawback of the system is that doing proposed exercises requires a fairly high level of skills and not all the tested can study the debugging process. In addition, training based on three exercises cannot show actual skills. There was no experiment to assess the effectiveness of the debugging training system. The eight most common errors of beginner developers were studied in article [12]. Eight short programs, each representing one type of a bug, were written, and the tested had to match the program with the type of the detected bug. The experiment involved 59 second-year students from Kent State University. The experiment was designed to track the order, in which a participant corrected errors, in addition to the time spent on each of the bugs. The results make it possible to identify the bugs that are more complex. However, they do not give an opportunity to assess the level of students' debugging skills and do not determine what methods and debugging tools were used.

The tools for computing thinking, one aspect of which is debugging, were developed in the study [13]. The Light-Bot (Canada) training game is used to explain its structure. The downside of Light-Bot is that it is used solely to teach programming and debugging skills, rather than analyze them. In addition, it has a very limited set of debugging tools and when moving to a professional IDE, it will be necessary to carry out the learning process again. The authors of research [14] collected and analyzed 450 bugs that students cannot solve and on which they turn to teachers for help. To collect the data, a web application, where students provide a brief description of an assignment and the problems they faced, was developed. After that, the teacher reviews a code with a student and helps him understand the problem. The most common types of bugs are discussed in class. The main result of the study is that approximately 22 % of problems are related to problem-solving skills, while the rest are associated with a combination of logical and syntax errors. Understanding the errors students face makes it possible to teach them the necessary debugging skills. Teachers use these data to continuously improve the syllabus. Information on the effectiveness of this approach is not given. The downside of this approach is that students should visit the site on their own and describe the problems they face, instead of collecting this information directly from the IDE.

In article [15], modern trends in software debugging methods were considered. A systematic debugging procedure was proposed in research [16]. The study shows that only a few students took a structured approach, but quickly returned to an unstructured one. Students were asked to find defects in a code analyze and fix them. In addition, students were asked to document their approach. As a result, in order to improve students' debugging skills, a systemic approach to debugging was developed. The drawback of this approach is that the conclusion about the skills is based on the approach documented by a student himself and manually.

Based on a study of the BlueJ-Blackbox dataset, the errors that Java beginners typically encounter are presented in paper [17]. Empirical studies show that semantic errors are more common than syntax ones, especially among experienced developers. Based on compilation events from 250,000 students from around the world, the debugging frequency and time were analyzed. The downside is that only compilation errors are analyzed and only from the BlueJ IDE, which is used to teach programming and is radically different from professional environments. As a result, after the transition to modern, professional IDE, it will be necessary to repeat the learning process.

Article [18] studied the debugging methods of experienced software developers through short interviews and observation of each of eight participants for several hours during one working day. The most common method among the respondents is "intuitive". They formulate hypotheses about a program and then do simple experiments to test them. Based on the obtained results, a questionnaire for a debugging survey was created. The results of the study are not subject to generalization. The main problem is a small scale, eight developers. Another problem is a limited period. The results of the study do not bear practical value for teaching students. In research [19], a study on how programmers set breakpoints was organized. Analyzing the operators, by which developers set breakpoints, it was found that 53 % of breakpoints were set in call operators and only 1 % in cycles. These data are also supported by study [20]. The conclusion of this study is that setting breakpoints and the step-by-step execution of a code are the most commonly used debugging methods.

As we can see, multifaceted research in debugging processes is not based on models of processes and does not make it possible to study the debugging process of each particular software developer.

Secondly, let us look at how the programming environment is used to study debugging processes.

A practical guide to the use of IDE is provided in research [21]. The research is based on the idea that the tools collecting data on the IDE use provide a more detailed understanding of the work of developers than it was previously possible. The paper describes approaches to mining data from different development environments, and some of these approaches are used in the developed extension. The dataset, which contains more than 600 hours of interaction between a programmer and the IDE, of which more than 26 hours are accompanied by computer screen video recordings and oral comments by developers, was presented in paper [22]. The drawback of this approach is that only events during code writing, without debugging it, were analyzed, and logged. In addition, no information is mined from the received videos, so their analysis can only be manual. The way users spend their time working at Microsoft Visual Studio was considered in [23]. Like in the developed extension, data on software development and debugging processes are extracted from the IDE. However, information on debugging is limited to breakpoints. In addition, the mined information is not analyzed in any way.

Hidden Markov Models (HMM) are also used as a means of mining developer's behavior from the data on the interaction with the IDE. According to this approach [24], a series of de-

bugging sessions, involving about 200 professional developers from ABB, Inc., was studied. The developed debugging model records the developers' behavior during setting breakpoints, beginning of debugging, and step-by-step execution of a code. The drawback of the proposed approach is that it collects data on only a few debugging tools and that it is semi-automatic. The HMM debugging process is manually constructed by an expert. In addition, only debugging data are extracted from the IDE without the development process. In paper [25], the paths chosen by students when debugging using the HMM were chosen. The downside of the work is that the described approach is only theoretical and there are no examples of its even potential use. In addition, the debugging process is considered very superficially, fixing only the run of a program in the debugging mode, without information about the tools used.

Swarm Debug Infrastructure (SDI), providing tools to collect, exchange and mine debug activities, was implemented [26]. Programmers can use the overall experience of previous debugging sessions. SDI was evaluated in an empirical experiment with 10 engineers. SDI is only designed to collect and share information about established breakpoints and debugging paths. It does not implement the analysis of the debugging process and the skills of developers. Paper [27] presents an online tool, Ladebug (Digital Equipment Corporation, USA), designed to support debugging skills training. In this tool, students use a systematic debugging process to identify and correct errors in preset exercises. The presented tool is used only to teach the basics of debugging and does not analyze the students' skills. In addition, the debugging process takes place inside a tool that is only used for learning with a very limited set of tools. When they move to a professional IDE, students will have to go through the learning process using actual, not educational tools. The cognitive processes of students while debugging applications that use eye-tracking were studied in article [28]. The students' eye movements were recorded to see how high- and low-efficiency students behave during debugging. The study found that beginners in programming followed a linear line-by-line approach when debugging computer programs, while students with prior programming experience followed a more logical and strategic approach. Thus, we decided to divide students into groups according to their success, creating an environment, in which they can think out loud. The downside of this approach is that it is impossible to analyze the debugging process during individual work.

At last, thirdly, explore how Process Mining tools are used.

Over the past decade, the application of the Process Mining has been effective in analyzing processes based on event data. The goal of Process Mining is to discover, monitor, and enhance processes by using the data from an event log from the information system [29]. The IEEE Task Force has released the Process Mining Manifesto [30]. The Manifesto was supported by 53 organizations and 77 Process Mining experts. It is aimed at promoting Process Mining. In addition, by determining a set of rules and listing critical issues, this manifesto should serve as a guide for software developers, scientists, and end-users.

Process Mining can be equally applied to software [31]. Using Process Mining methods, in paper [32], they studied how programmers interact with software repositories. In study [33], the records in issue tracking systems were examined. The general drawback of these papers is that the data are extracted from the systems that do not make it possible to assess the student's contribution to the result because they do not provide information about the process of software writing and debugging. In article [34], the compliance check was used to test the behavior of developers. The actions performed by 40 beginners-developers executing coding actions in five development sessions were assessed. The work mainly focuses on using a compliance-based approach, comparing the execution of processes recorded in event logs with some of the expected behaviors presented as a process model. The disadvantage of the proposed approach is that the development and the debugging process are seen as a single process. A similar approach was introduced in research [35], but there was only a basic concept, leaving its implementation and verification for subsequent work.

The systematization of research results suggests a lack of knowledge of how programmers fix problems in the software debugging process. All of this makes it possible to argue that a study on the development of tools for tracking, modeling, and analyzing software debugging processes is appropriate.

3. The aim and objectives of the study

The aim of this study is to develop tools to monitor and analyze software debugging processes. By analyzing how developers use the IDE, one can discover the patterns of programmers' behavior during debugging and identify the problems they face.

To accomplish the goal, the following tasks were set:

– to develop a constructor to form an event log that displays the debugging process;

– to develop the tools to record debugging events from the developer's IDE;

– to form the model of a debugging process using the Process Mining methods.

4. Construction of an event log with data on the IDE use during debugging

Constructive-synthesizing modeling (CSM) was applied to formalize the process of collecting data on the IDE use during debugging. The basics of the CSM are shown in papers [37–40]. The CSM can be used to model and formalize any structures and constructive processes. The CSM tools were used to solve problems such as:

– creating the timetable of a university course [41];

– simulating lightning flashes in the thunderstorm front [42];

– representation of geometrical fractals [43];

– modeling the adaptation of compression algorithms [44];

– formalization and automation of the process of documents' comparison to detect text borrowings [45] and many others.

A wide range of these tasks demonstrates the versatility and prospects of using the CSM to address the challenges of various subject areas. These works reveal the versatility and high commonality of this modeling method.

The first stage of development is the specialization of the generalized constructor [37]. Specialization determines the semantic nature of the carrier, the purpose of the construction, the final set of operations, their semantics and attributes, the order of execution, and limitations [37, 38].

In an informal form, the ontology of the generalized constructor is presented in papers [37, 38]. The main provisions of the ontological accompaniment of the CSM are presented

in articles [46, 47]. The work presents only those components that are necessary for further presentation.

The goal of a constructor is to create an event log that displays the debugging process.

Initial conditions are non-terminal σ, from which the inference begins.

Completion conditions – all events were processed.

Specialization of the generalized constructor:

$$C=\langle \mathrm{M},\Sigma,\Lambda\rangle \ {}_{S}\mapsto C_L=\langle \mathrm{M}_L,\Sigma_L,\Lambda_L\rangle, \quad (1)$$

where ${}_{S}\mapsto$ is the specialization operation (performed by an external executor), M_L is the heterogeneous replenishable carrier, including a set of terminals and non-terminals, $\sum_L$ is the signature of relations and corresponding operations, Λ_L is the information support of construction.

Terminals and their attributes:

– ${}_{id,\,sdt,\,fdt,\,ss,\,el,\,d,\,p}s$ is the debugging session, *id* is its identifier, *sdt*, *fdt* are the time of its beginning and ending, *ss* is the array of snapshots (snapshot is a file of a program code at the moment of time) at the beginning of a session, *el* is the array of events during the debugging session, *d* is the information about a developer, *p* is the information about a project;

– attributes $el\left({}_{e\ index,l}el\right)$: *index* is the index of event *e* in an array, *l* is the array size;

– ${}_{id,\,t,\,cdt,\,fp,\,ln,\,context}e$ are the events during the debugging session, *id* is the identifier, *t* is the type, *cdt* is the time of event emerging, *fp* is the path to a debugged file, *ln* is the line of the code, with which an event is connected, *context* is the event context, an object of the dynamic structure with information about the event environment (the IDE, project, developer, file), the object structure can be different for various events;

– attributes d (${}_{id,\ name}d$): *id* is the identifier, *name* is the developer's name;

– attributes p (${}_{id,\ name}p$): *id* is the identifier, *name* is the project name;

– ${}_{context}sdse$ is the event of debugging run from an external executor (IDE);

– ${}_{context}de$ is the debugging event from the IDE;

– ${}_{context}edse$ is the event of debugging completion from the IDE;

– ${}_{traces}log$ is the file of an event log in eXtensible Event Stream format, *traces* is the array of sequences (events) with attributes $\left({}_{trace\ index,l}traces\right)$: *index* is the index of *trace* sequence, *l* is the size of the *traces* array;

– ${}_{id,\,sdt,\,fdt,\,p,\,d,\,events}\,trace$ is the array of events at a single process execution, *events* is the array of events during the debugging session.

The constructor has the following operation over the attributes:

– ∘(*t*) is assigning the value to terminal *t* by an external executor;

– ≺(*sdse, s*) is the creation of debugging session *s* on event *sdse*;

– ≻ (*s, de*) is addition of event *de* to debugging session *s*;

– ≡ (*s, edse*) is the completion of the debugging session *s* on event *edse*;

– ◁ (*did, pid, log*) is the creation of the file of an event log *log* for project *pid* and developer *did*;

– ▷ (*de, s, log*) is the transfer of event *de* from debugging session *s* to the file of event log *log*;

– ≈(*log*) is saving a file of an event log.

Signature $\Sigma_L=\langle \Xi,\Theta,\Phi,\{\rightarrow,\hookleftarrow\},\Psi\rangle$ contains a set of operations and relations, where $\Xi\supset\{\bullet,:\}$ is the operation and transformation of the carrier's elements, $\Theta=\{\Rightarrow,|\Rightarrow,\|\Rightarrow\}$ is the substitution and output operations, Φ are the operations over attributes, as well as relations of substitution $(\rightarrow)$ and attribution $(\hookleftarrow)$, $\Psi=\{\psi_i:\langle s_i,g\rangle_i\}$ is the set of substitution rules, s_i is the sequence of substitution relations, g_i is the sequence of operations over attributes.

Interpretation is linking the algorithms implementing a certain algorithmic structure (algorithm constructor) with signature operations. In the interpretation process, the models of the constructor and the internal performer are related. The result is a constructive system that is capable and has construction tools [37, 38].

To interpret C_L, it is necessary to refine the basic algorithmic structure (BAS) [37, 38].

Assume that there is the following BAS:

$$C_{A,L}=\langle \mathrm{M}_{A,L},V_{A,L},\Sigma_{A,L},\Lambda_{A,L}\rangle, \quad (2)$$

where $V_{A,L}$ is the finite set of basic algorithms $A_i|_{X_i}^{Y_i}$ of an internal construction executor with a set of input and output data X_i и Y_i.

The following algorithms implement operations over attributes:

$$A_1|_{t}^{t},\ A_2|_{s,sdse}^{s},\ A_3|_{s,de}^{s},\ A_4|_{s,edse}^{s},$$

$$A_5|_{did,pid,\log}^{\log},\ A_6|_{de,s,\log}^{\log},\ A_7|_{\log}^{\log}. \quad (3)$$

Create a constructive system:

$$\left\langle \begin{array}{l} C_L=\langle M_L,\Sigma_L,\Lambda_L\rangle, \\ C_{A,L}=\langle M_{A,L},V_{A,L},\Sigma_{A,L},\Lambda_{A,L}\rangle \end{array}\right\rangle,$$

$${}_{I}\mapsto C_{I,L}=\langle M_{I,L},\Sigma_{I,L},\Lambda_{I,L}\rangle,$$

$$\Lambda_{I,L}=\Lambda_L\cup\left\{\begin{array}{l} \left(A_1|_{t}^{t}\hookleftarrow\circ\right),\left(A_2|_{s,sdse}^{s}\hookleftarrow\prec\right),\left(A_3|_{s,de}^{s}\hookleftarrow\succ\right), \\ \left(A_4|_{s,edse}^{s}\hookleftarrow\equiv\right),\left(A_5|_{did,pid,\log}^{\log}\hookleftarrow\lhd\right), \\ \left(A_6|_{de,s,\log}^{\log}\hookleftarrow\rhd\right),\left(A_7|_{\log}^{\log}\hookleftarrow\approx\right) \end{array}\right\}, \quad (4)$$

where ${}_{I}\mapsto$ is the interpretation operation.

The specification of the constructor implies determining specific rules of substitution, limitations, initial conditions, and conditions of construction completion, basic elements of a carrier with their properties, and property values. After the interpretation and specification operations performed by an external executor, the constructive system has everything necessary for the autonomous creation of structures [37–40].

Specify constructor C_L to create a debugging log:

$$C_{I,L}=\langle M_{I,L},\Sigma_{I,L},\Lambda_{I,L}\rangle_K\mapsto C_L=\langle M_{K,L},\Sigma_{K,L},\Lambda_{K,L}\rangle\rangle, \quad (5)$$

where ${}_{K}\mapsto$ is the specification operation.

The substitution rules are presented below (6) to (9).

Rule s_1 is applied if event *sdse* was obtained from an external executor. As a result of following the rule, debugging session *s* is created

$$s_1=\langle\sigma\rightarrow(sdse\bullet\alpha)\rangle,\quad g_1=\langle\circ(sdse),\prec(sdse,s)\rangle. \quad (6)$$

Rule s_2 is applied if the event *de* is obtained. As a result, event *de* is added to debugging session *s*

$$s_2 = \langle \alpha \to de \bullet \alpha \rangle, \quad g_2 = \langle \circ(de), \succ(s, de) \rangle. \tag{7}$$

Rule s_3 is applied if event *edse* is obtained. As a result, debugging session *s* is closed

$$s_3 = \langle \alpha \to edse \bullet \beta \rangle, \quad g_3 = \langle \circ(edse), \cong(s, edse) \rangle. \tag{8}$$

As a result of keeping to rule s_4, an event log file *log* is created. The file will be filled with the events from the previous debugging sessions of a developer for this project or it is empty and is filled from the current debugging session

$$s_4 = \langle \beta \to log \rangle,$$

$$g_4 = \langle \triangleleft(id \lrcorner d \lrcorner s, id \lrcorner p \lrcorner s, log), \triangleright(de, s, log) \rangle. \tag{9}$$

The implementation, carried out by an internal executor of the system, consists of the formation of a structure of carriers-elements by performing the algorithms related to substitution operations. Only the constructor, which was previously specialized, interpreted, and specified, can be implemented. The result is the formation of a debugging log file.

5. Development of the tools to record debugging events from the developer's IDE

Based on the constructive model, an extension for Microsoft Visual Studio, in which all debugging activities are recorded in event logs, was developed. To save and control the tracked events, a data model that matches previously presented terminals with their attributes was created. Fig. 1 shows the proposed model as a class diagram.

The class diagram is based on the event model of the debugging process. Events occur when a programmer performs some actions during a debugging session. They contain the type, time, context, and path to a debugging file with the line number on which the event occurred, if necessary. We save not only simple information about executed commands, but also specific data (a dynamic structure object with information about the event environment, context), depending on the type of event. *Developer* is the user who runs and performs debugging sessions. *Project* is a solution from a set of components. *Session* is a debugging session. It contains information about a developer, a project, and debugging events. Each event is related to a specific type. Each event type is represented by a specific class, which is the implementation of the abstract basic class *Event*. The *EventType* listing contains all possible types of events that are being tracked.

Fig. 2 shows the architecture of a top-level, developed software tool.

Fig. 2 shows two main components: Debug Event Tracker and Debug Event Analyzer. Debug Event Tracker is an extension to Microsoft Visual Studio, written in C#. The component is responsible for collecting data from the current project debugging sessions and sending them to a remote server. The extension not only records what events occur in the IDE but also stores relevant contextual information about them. Debug Event Analyzer is responsible for analyzing debugging operations using the Process Mining methods.

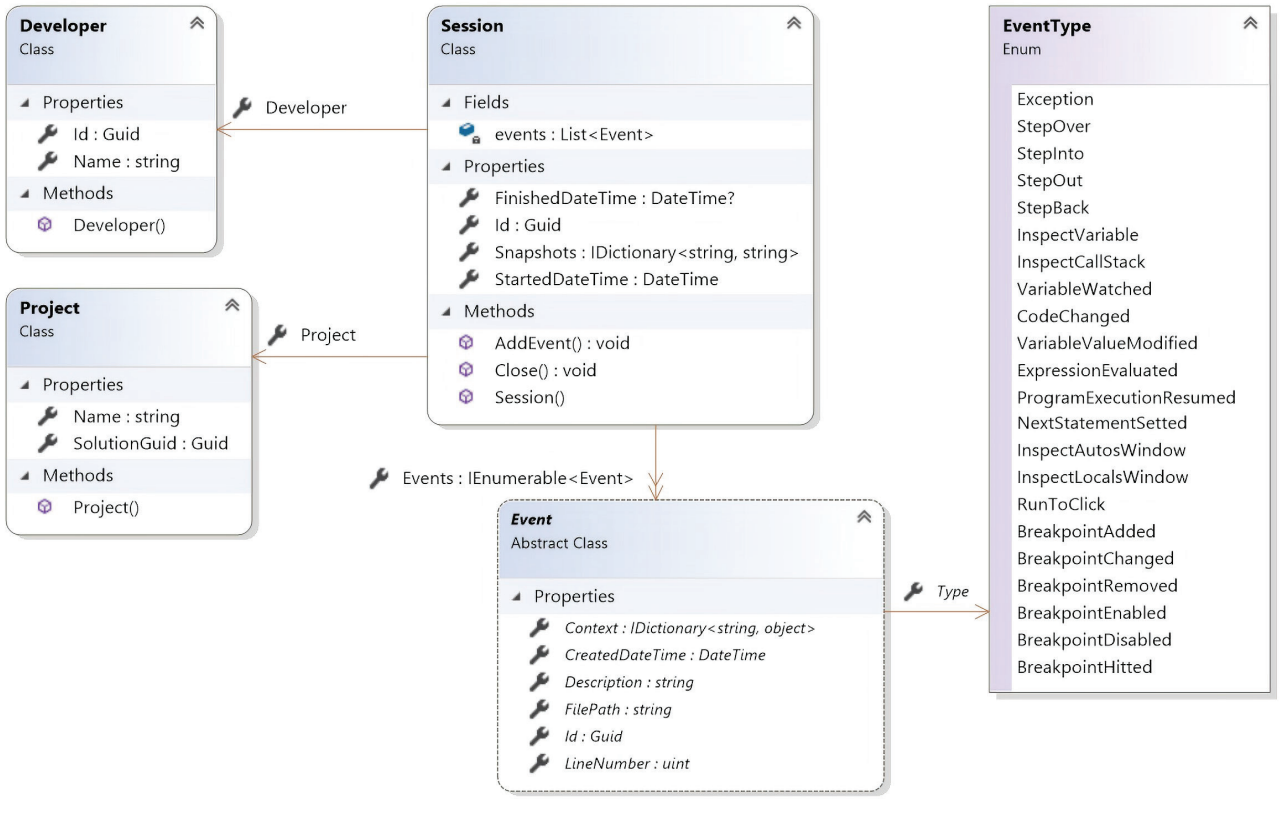

Fig. 1. Diagram of classes of the model of data for saving and controlling event tracking

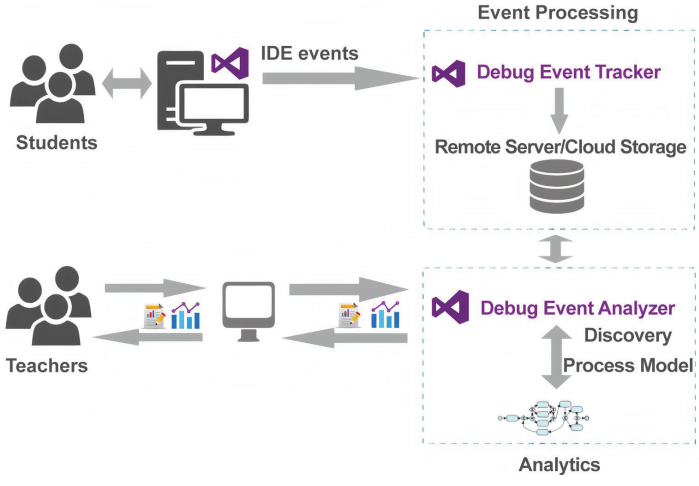

Fig. 2. The architecture of the tool for debugging process monitoring and analysis

6. Processing an event log using the Process Mining methods

Process Mining purposes to discover, check, and enhance actual processes by mining the information from event logs that ensure understanding of the process. The result of Process Mining can range from a complete model of a process to the most frequent process paths or deviations.

The starting point of Process Mining is an event log. Each event in such a log refers to an action that can be performed on a resource at a specific time and for a particular session. An event log is usually structured as a set of traces, where each trace makes a chain of actions created by a single process run (session). As a minimum, event recording includes an identifier of the process session, to which an event, time, and a series of additional attributes are applied. A description of the attributes of an event log is shown in Table 1.

An event log is formed in the eXtensible Event Stream (XES) format [36], which is the standard format for Process Mining developed by the IEEE working group to record events.

The first method of Process Mining is discovery. The discovery method takes a log of events, consisting of recording all the actions that occur during a software debugging process, and forms a model that represents how and in what order the process was executed. ProM is used to discovery a debugging process model from an event log. ProM is a common open source framework, the de-facto standard for Process Mining implementation [48].

Table 1

Attributes of events

Attribute	Level	Description
Name	Trace	Identifier of debugging session
StartedDateTime	Trace	Time of beginning the debugging session
FinishedDateTime	Trace	Time of finishing the debugging session
Project	Trace	Project identifier
Developer	Trace	Developer's identifier
Activity	Event	Event name
Timestamp	Event	Time of event occurrence
Context	Event	Event context
Resource	Event	Path to debugged file
LineNumber	Event	Number of line, to which an event is related

The event logs of five debugging sessions in the XES format, tracked with the use of the IDE, and during the software debugging process were imported into the ProM to show in detail how the debugging process was performed. Fig. 3 presents the traces, where each trace represents one debugging session.

The structures of representation of the debugging process, ensuring a reciprocal unambiguous match (representation and process) for the first two traces (10) to (11).

$$\sigma \overset{s_1}{\Rightarrow} sdse \bullet \alpha \overset{s_3}{\Rightarrow} sdse \bullet edse \bullet \beta \overset{s_4}{\Rightarrow} sdse \bullet edse \bullet log, \quad (10)$$

$$\sigma \overset{s_1}{\Rightarrow} sdse \bullet \alpha \overset{s_2}{\Rightarrow} sdse \bullet de \bullet \alpha \overset{s_3}{\Rightarrow} sdse \bullet de \bullet edse \bullet \beta \overset{s_4}{\Rightarrow} sdse \bullet de \bullet edse \bullet log. \quad (11)$$

ProM's Inductive Miner Visual [49], which represents the direct-follow graphs (DFG), was used to form a debugging process model. The DFG represents the actions in the form of rectangles and relations of actions if one of them directly follows the other. In addition, each edge has a weight indicating the number of entries in the event log. The result of this stage is the process model shown in Fig. 4.

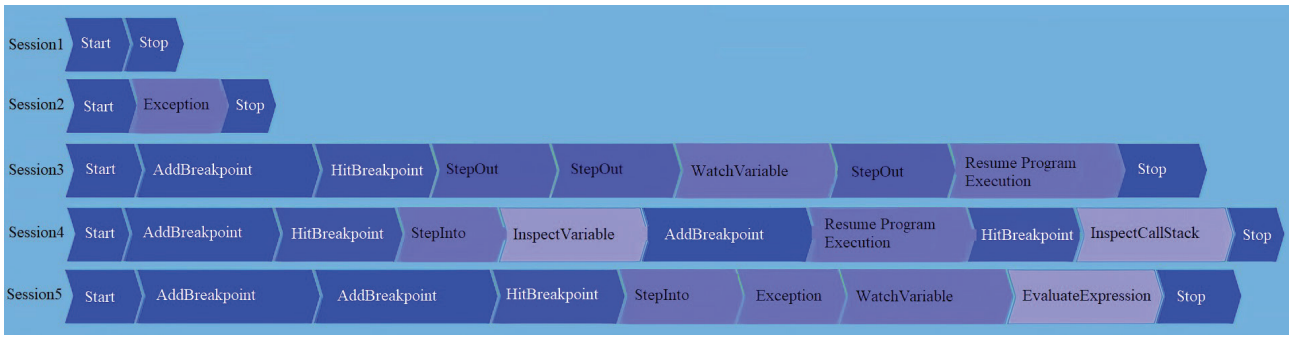

Fig. 3. Traces

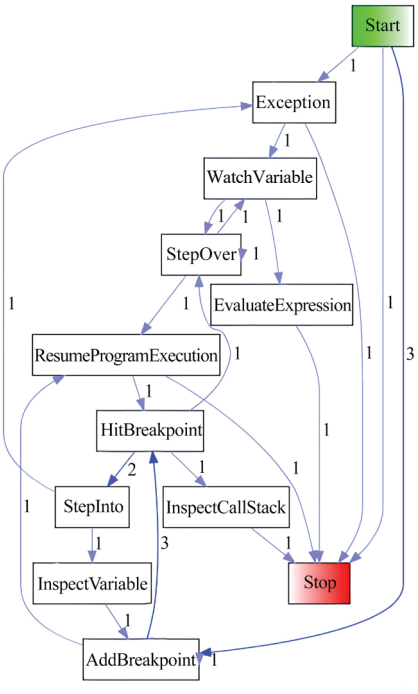

Fig. 4. The debugging process model discovered using Inductive Visual Miner

The second method of Process Mining is the conformance checking. By compliance checking, one can match different process implementations and find out behavioral similarities and differences. In the study, we conduct the conformance checking twice: when the detected model is reproduced on an event log and when measuring the deviation between a predetermined model and a detected model. The fitness feature [30] is used to match the check. A model has excellent suitability if all the tracks can be reproduced by a model from start to finish. Fitness is characterized by the number from 0 (very bad) to 1 (excellent).

A predetermined model shows which transitions between actions are possible (Fig. 5).

The ProM toolkit contains plug-ins that makes it possible to calculate suitability by a model and by an event log file, as well as suitability by two models. Conformance checking are performed using the Visualize deviations ProM plug-in [49]. The suitability of a log file, formed using the extension to Visual Studio, in relation to the resulting model is equal to unity, which proves reliability and adequacy of the resulting model.

Fig. 5. Admissible transitions

7. Discussion of results of studying the developed tools for analyzing software debugging processes

Three components of the research were developed:

– an extension to Visual Studio that enables getting information about events when coding and debugging programs. Visual Studio environment offers the opportunity of getting information about the programmer's actions linked to the program's text. The functionality of the plug-in, which enables gathering information about the debugging process, is shown in Fig. 2;

– the expansion of plug-in functionality associated with linking events to software essences became possible based on the formalism of the constructor (1) to (9);

– Process Mining (ProM) tools and methods for tracking and analyzing processes. The formation of a model of a particular process is shown in Fig. 3, 4. The analysis is based on a generalized process model as shown in Fig. 5.

Several approaches [8–28] are used to improve debugging and debugging training processes. Only a part of them [21–28] are based on objective information from software development and debugging tools. The process is analyzed manually. As a part of this study, it was possible to automate the analysis of the debugging process by using the known tools and methods of Process Mining. Match checks are performed using the Visualize deviations ProM plug-in.

The results of the study, which analyzed the event logs of five debugging sessions using the Process Mining toolkit, were presented. After applying the Process Mining methods to event logs, we obtained a model of the software debugging process. The results show that Process Mining methods are useful for understanding how programmers perform debugging activities and what difficulties they typically face.

Thus, by developing the tools and methods of collecting information on debugging processes proposed in [21–28] and by automating the formation, verification, and analysis of process models, a complete technological cycle of process quality assessment was developed. This study enables both a teacher and a trained programmer to obtain additional information about the debugging process and its quality, thereby expanding the capability to effectively manage the debugging process and improve skills. As it is known, the more information is known about the process, the more effective management can be.

The results show that Process Mining methods are useful to understand how programmers perform debugging activities and what difficulties they typically face.

Based on the research results, adaptive training methods can be applied for students with varying degrees of academic performance.

Naturally, the proposed approach is not universal. Thus far, it only applies to a specific development environment – Visual Studio. For other development environments, only the extension needs to be improved.

This article is the first step to understanding debugging skills. The ultimate goal is to improve developers' debugging skills. In the future, it is supposed to develop a recommendation system for teachers and those learning the software debugging processes.

8. Conclusions

1. A constructive and production modeling approach is used to formalize the process of collecting data on the use of the IDE during debugging. A constructor aimed at the creation of a file of a log of debugging activity in the XES format was developed.

2. Based on the constructive model, an extension for Microsoft Visual Studio, in which all debugging activities are recorded in event logs, was developed. Analyzing the interaction between programmers and the IDE helps researchers interpret the behavior of developers.

3. The Process Mining methods were used to construct a model of the debugging process and validate it. The experiment was conducted using an open code platform ProM. The results showed that the software debugging process could be effectively analyzed using the Process Mining approach.

References

1. IEEE Standard Glossary of Software Engineering Terminology (1990). doi: https://doi.org/10.1109/ieeestd.1990.101064
2. LaToza, T. D., Myers, B. A. (2010). Developers ask reachability questions. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10. doi: https://doi.org/10.1145/1806799.1806829
3. Shynkarenko, V., Zhevago, O. (2019). Visualization of program development process. 2019 IEEE 14th International Conference on Computer Sciences and Information Technologies (CSIT). doi: https://doi.org/10.1109/stc-csit.2019.8929774
4. Shynkarenko, V., Zhevaho, O. (2020). Constructive modeling of the software development process for modern code review. In IEEE 2020 15th International Scientific and Technical Conference on Computer Sciences and Information Technologies, CSIT 2020.
5. Denny, P., Luxton-Reilly, A., Tempero, E., Hendrickx, J. (2011). Understanding the syntax barrier for novices. Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education - ITiCSE '11. doi: https://doi.org/10.1145/1999747.1999807
6. Denny, P., Luxton-Reilly, A., Carpenter, D. (2014). Enhancing syntax error messages appears ineffectual. Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education - ITiCSE '14. doi: https://doi.org/10.1145/2591708.2591748
7. Pegoraro, M., van der Aalst, W. M. P. (2019). Mining Uncertain Event Data in Process Mining. 2019 International Conference on Process Mining (ICPM). doi: https://doi.org/10.1109/icpm.2019.00023
8. Bers, M. U., Flannery, L., Kazakoff, E. R., Sullivan, A. (2014). Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. Computers & Education, 72, 145–157. doi: https://doi.org/10.1016/j.compedu.2013.10.020
9. Lee, G. C., Wu, J. C. (1999). Debug It: A debugging practicing system. Computers & Education, 32 (2), 165–179. doi: https://doi.org/10.1016/s0360-1315(98)00063-3
10. Maalej, W., Tiarks, R., Roehm, T., Koschke, R. (2014). On the Comprehension of Program Comprehension. ACM Transactions on Software Engineering and Methodology, 23 (4), 1–37. doi: https://doi.org/10.1145/2622669

11. Yamamoto, R., Noguchi, Y., Kogure, S., Yamashita, K., Konishi, T., Itoh, Y. (2016). Design of a learning support system and lecture to teach systematic debugging to novice programmers. In ICCE 2016 – 24th International Conference on Computers in Education: Think Global Act Local – Main Conference Proceedings, 276–281.
12. Alqadi, B. S., Maletic, J. I. (2017). An Empirical Study of Debugging Patterns Among Novices Programmers. Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education. doi: https://doi.org/10.1145/3017680.3017761
13. Gouws, L. A., Bradshaw, K., Wentworth, P. (2013). Computational thinking in educational activities. Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '13. doi: https://doi.org/10.1145/2462476.2466518
14. Bryce, R. C., Cooley, A., Hansen, A., Hayrapetyan, N. (2010). A one year empirical study of student programming bugs. 2010 IEEE Frontiers in Education Conference (FIE). doi: https://doi.org/10.1109/fie.2010.5673143
15. Ghosh, D., Singh, J. (2019). A Systematic Review on Program Debugging Techniques. Smart Computing Paradigms: New Progresses and Challenges, 193–199. doi: https://doi.org/10.1007/978-981-13-9680-9_16
16. Bottcher, A., Thurner, V., Schlierkamp, K., Zehetmeier, D. (2016). Debugging students' debugging process. 2016 IEEE Frontiers in Education Conference (FIE). doi: https://doi.org/10.1109/fie.2016.7757447
17. Altadmri, A., Brown, N. C. C. (2015). 37 Million Compilations. Proceedings of the 46th ACM Technical Symposium on Computer Science Education - SIGCSE '15. doi: https://doi.org/10.1145/2676723.2677258
18. Perscheid, M., Siegmund, B., Taeumel, M., Hirschfeld, R. (2016). Studying the advancement in debugging practice of professional software developers. Software Quality Journal, 25 (1), 83–110. doi: https://doi.org/10.1007/s11219-015-9294-2
19. Petrillo, F., Mandian, H., Yamashita, A., Khomh, F., Gueheneuc, Y.-G. (2017). How Do Developers Toggle Breakpoints? Observational Studies. 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). doi: https://doi.org/10.1109/qrs.2017.39
20. Beller, M., Spruit, N., Spinellis, D., Zaidman, A. (2018). On the dichotomy of debugging behavior among programmers. Proceedings of the 40th International Conference on Software Engineering. doi: https://doi.org/10.1145/3180155.3180175
21. Snipes, W., Murphy-Hill, E., Fritz, T., Vakilian, M., Damevski, K., Nair, A. R., Shepherd, D. (2015). A Practical Guide to Analyzing IDE Usage Data. The Art and Science of Analyzing Software Data, 85–138. doi: https://doi.org/10.1016/b978-0-12-411519-4.00005-7
22. Yamashita, A., Petrillo, F., Khomh, F., Guéhéneuc, Y.-G. (2018). Developer interaction traces backed by IDE screen recordings from think aloud sessions. Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18. doi: https://doi.org/10.1145/3196398.3196457
23. Bellman, C., Seet, A., Baysal, O. (2018). Studying developer build issues and debugger usage via timeline analysis in visual studio IDE. Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18. doi: https://doi.org/10.1145/3196398.3196463
24. Damevski, K., Chen, H., Shepherd, D., Pollock, L. (2016). Interactive exploration of developer interaction traces using a hidden Markov model. Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16. doi: https://doi.org/10.1145/2901739.2901741
25. Piech, C., Sahami, M., Koller, D., Cooper, S., Blikstein, P. (2012). Modeling how students learn to program. Proceedings of the 43rd ACM Technical Symposium on Computer Science Education - SIGCSE '12. doi: https://doi.org/10.1145/2157136.2157182
26. Petrillo, F., Soh, Z., Khomh, F., Pimenta, M., Freitas, C., Gueheneuc, Y.-G. (2016). Understanding interactive debugging with Swarm Debug Infrastructure. 2016 IEEE 24th International Conference on Program Comprehension (ICPC). doi: https://doi.org/10.1109/icpc.2016.7503740
27. Luxton-Reilly, A., McMillan, E., Stevenson, E., Tempero, E., Denny, P. (2018). Ladebug: an online tool to help novice programmers improve their debugging skills. Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE 2018. doi: https://doi.org/10.1145/3197091.3197098
28. Lin, Y.-T., Wu, C.-C., Hou, T.-Y., Lin, Y.-C., Yang, F.-Y., Chang, C.-H. (2016). Tracking Students' Cognitive Processes During Program Debugging – An Eye-Movement Approach. IEEE Transactions on Education, 59 (3), 175–186. doi: https://doi.org/10.1109/te.2015.2487341
29. Van der Aalst, W. (2012). Process Mining. ACM Transactions on Management Information Systems, 3 (2), 1–17. doi: https://doi.org/10.1145/2229156.2229157
30. Van der Aalst, W., Adriansyah, A., de Medeiros, A. K. A., Arcieri, F., Baier, T., Blickle, T. et. al. (2012). Process Mining Manifesto. Lecture Notes in Business Information Processing, 169–194. doi: https://doi.org/10.1007/978-3-642-28108-2_19
31. Rubin, V. A., Mitsyuk, A. A., Lomazova, I. A., van der Aalst, W. M. P. (2014). Process mining can be applied to software too! Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '14. doi: https://doi.org/10.1145/2652524.2652583
32. Poncin, W., Serebrenik, A., Brand, M. van den. (2011). Process Mining Software Repositories. 2011 15th European Conference on Software Maintenance and Reengineering. doi: https://doi.org/10.1109/csmr.2011.5
33. Sebu, M. L., Ciocarlie, H. (2014). Applied process mining in software development. 2014 IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI). doi: https://doi.org/10.1109/saci.2014.6840098
34. Ardimento, P., Bernardi, M. L., Cimitile, M., Maggi, F. M. (2019). Evaluating Coding Behavior in Software Development Processes: A Process Mining Approach. 2019 IEEE/ACM International Conference on Software and System Processes (ICSSP). doi: https://doi.org/10.1109/icssp.2019.00020

35. Caldeira, J., Abreu, F. B. e. (2016). Software Development Process Mining: Discovery, Conformance Checking and Enhancement. 2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC). doi: https://doi.org/10.1109/quatic.2016.061
36. Verbeek, H. M. W., Buijs, J. C. A. M., van Dongen, B. F., van der Aalst, W. M. P. (2011). XES, XESame, and ProM 6. Lecture Notes in Computer Science, 60–75. doi: https://doi.org/10.1007/978-3-642-17722-4_5
37. Shynkarenko, V. I., Ilman, V. M. (2014). Constructive-Synthesizing Structures and Their Grammatical Interpretations. I. Generalized Formal Constructive-Synthesizing Structure. Cybernetics and Systems Analysis, 50 (5), 655–662. doi: https://doi.org/10.1007/s10559-014-9655-z
38. Shynkarenko, V. I., Ilman, V. M. (2014). Constructive-Synthesizing Structures and Their Grammatical Interpretations. II. Refining Transformations. Cybernetics and Systems Analysis, 50 (6), 829–841. doi: https://doi.org/10.1007/s10559-014-9674-9
39. Shynkarenko, V. I., Ilman, V. M., Skalozub, V. V. (2009). Structural models of algorithms in problems of applied programming. I. Formal algorithmic structures. Cybernetics and Systems Analysis, 45 (3), 329–339. doi: https://doi.org/10.1007/s10559-009-9118-0
40. Shynkarenko, V. I., Ilman, V. M., Skalozub, V. V. (2009). Structural models of algorithms in problems of applied programming. II. Structural-algorithmic approach to software simulation. Cybernetics and Systems Analysis, 45 (4), 544–550. doi: https://doi.org/10.1007/s10559-009-9122-4
41. Shinkarenko, V. I., Zhevago, O. O. (2019). Generating university course timetable using constructive modeling. Radio Electronics, Computer Science, Control, 3, 152–162. doi: https://doi.org/10.15588/1607-3274-2019-3-17
42. Shynkarenko, V., Lytvynenko, K., Chyhir, R., Nikitina, I. (2019). Modeling of Lightning Flashes in Thunderstorm Front by Constructive Production of Fractal Time Series. Advances in Intelligent Systems and Computing, 173–185. doi: https://doi.org/10.1007/978-3-030-33695-0_13
43. Shynkarenko, V. I. (2019). Constructive-Synthesizing Representation of Geometric Fractals. Cybernetics and Systems Analysis, 55 (2), 186–199. doi: https://doi.org/10.1007/s10559-019-00123-w
44. Shynkarenko, V. I., Vasetska, T. M. (2015). Modeling the Adaptation of Compression Algorithms by Means of Constructive-Synthesizing Structures. Cybernetics and Systems Analysis, 51 (6), 849–862. doi: https://doi.org/10.1007/s10559-015-9778-x
45. Kuropiatnyk, O., Shynkarenko, V. (2020). Text borrowings detection system for natural language structured digital documents. In CEUR Workshop Proceedings, 2604, 294–305.
46. Skalozub, V., Ilman, V., Shynkarenko, V. (2017). Development of ontological support of constructive-synthesizing modeling of information systems. Eastern-European Journal of Enterprise Technologies, 6 (4 (90)), 58–69. doi: https://doi.org/10.15587/1729-4061.2017.119497
47. Skalozub, V., Ilman, V., Shynkarenko, V. (2018). Ontological support formation for constructive-synthesizing modeling of information systems development processes. Eastern-European Journal of Enterprise Technologies, 5 (4 (95)), 55–63. doi: https://doi.org/10.15587/1729-4061.2018.143968
48. Van der Aalst, W. (2016). Process mining: Data science in action. Springer. doi: https://doi.org/10.1007/978-3-662-49851-4
49. Leemans, S. J. J., Fahland, D., van der Aalst, W. M. P. (2018). Scalable process discovery and conformance checking. Software & Systems Modeling, 17 (2), 599–631. doi: https://doi.org/10.1007/s10270-016-0545-x