UDC 519.2

O. I. ZINKOVSKYI, R. O. GAMZAYEV, A. BOLLIN, M. V. TKACHUK

A FUZZY-BASED APPROACH TO AUTOMATED DEFECT IDENTIFICATION IN DISTRIBUTED SOFTWARE SYSTEMS AND SOFTWARE PRODUCT LINES

An approach to the improvement of the efficiency of the bug tracking process in distributed software systems and software product lines via automated identification of duplicate report groups and report groups collected from correlated bugs, combined with bug localization within a software product line is considered. A brief overview of the problem of automated report collection and aggregation is made, several existing software tools and solutions for report management and analysis are reviewed, and basic functionality of a typical report management system is identified. In addition to this, a concept of a report correlation group is introduced and an automated crash report aggregation method based on the rules for comparison of crash signatures, top frames, and frequent closed ordered sub-sets of frames of crash reports is proposed. To evaluate these rules, two separate fuzzy models are built, the first one to calculate the output of the Frequent Closed Ordered Sub-Set Comparison rule, and the second one to interpret and combine the output of all three rules and produce an integrated degree of crash report's similarity to an existing report correlation group or to another report. A prototype of a report management system with report aggregation capabilities is developed and tested using imported from the publicly available Mozilla Crash Stats project report groups. During the experiment, a precision of 90% and a recall of 81% are achieved. Lastly, an approach to localize the largest identified report groups and represented by them bugs within a concrete software product line based on an information basis consisting of a feature model, a list of software components, and a mapping between features and components is proposed, conclusions are drawn, and goals for the future work are outlined.

Keywords: crash reports, automated crash report collection and aggregation, bug localization, fuzzy logic, distributed software systems, software product lines, Mozilla Crash Stats project, Socorro, report management system, bug tracking.

О. І. ЗІНЬКОВСЬКИЙ, Р. О. ГАМЗАЄВ, А. БОЛЛІН, М. В. ТКАЧУК

ПІДХІД З ВИКОРИСТАННЯМ НЕЧІТКОЇ ЛОГІКИ ДО АВТОМАТИЗОВАНОЇ ІДЕНТИФІКАЦІЇ ДЕФЕКТІВ В РОЗПОДІЛЕНИХ ПРОГРАМНИХ СИСТЕМАХ ТА ЛІНІЙКАХ ПРОГРАМНИХ ПРОДУКТІВ

Розглянуто підхід до підвищення ефективності процесу відстеження помилок в розподілених програмних системах та лінійках програмних продуктів шляхом автоматизованої ідентифікації дубльованих груп звітів та груп звітів, зібраних з корельованих помилок, у поєднанні з локалізацією помилок серед компонентів лінійок програмних продуктів. Зроблено короткий огляд проблеми автоматизованого збору та агрегації звітів, розглянуто кілька існуючих програмних засобів для аналізу звітів, а також визначено основні функціональні можливості типової системи управління звітами. Крім того, запропоновано концепцію кореляційної групи звітів та наведено автоматизований метод агрегації звітів, який базується на правилах порівняння підписів звітів, верхньої форми звітів, та трасувальних стеків звітів про збої. Для оцінки цих правил будуються дві окремі нечіткі моделі – перша для розрахунку результату правила порівняння трасувальних стеків звітів, а друга - для інтерпретації та поєднання результатів усіх трьох правил і створення інтегрованого ступеня подібності звіту про збій з існуючою кореляційною групою звітів або іншим звітом. За допомогою імпорту груп звітів з загальнодоступного репозиторію Mozilla, тестується розроблений прототип системи управління та агрегації звітів. Під час експерименту досягається точність в 90% і повнота в 81%. Нарешті, пропонується підхід до локалізації найбільших ідентифікованих груп звітів та представлених ними помилок у лінійці програмних продуктів на основі інформаційної бази, що складається з функціональної моделі, списку програмних компонентів та взаємозв'язків між функціями та компонентами, робляться висновки та визначаються цілі для подальшої роботи.

Ключові слова: звіти про збої, автоматизований збір та агрегація звітів, локалізація дефектів, нечітка логіка, розподілені програмні системи, лінійки програмних продуктів, проект Mozilla Crash Stats, Socorro, система управління звітами, відстеження помилок.

А. И. ЗИНЬКОВСКИЙ, Р. А. ГАМЗАЕВ, А. БОЛЛИН, Н. В. ТКАЧУК

ПОДХОД С ИСПОЛЬЗОВАНИЕМ НЕЧЁТКОЙ ЛОГИКИ ДЛЯ АВТОМАТИЗИРОВАННОЙ ИДЕНТИФИКАЦИИ ДЕФЕКТОВ В РАСПРЕДЕЛЁННЫХ ПРОГРАММНЫХ СИСТЕМАХ И ЛИНЕЙКАХ ПРОГРАММНЫХ ПРОДУКТОВ

Рассмотрен подход к повышению эффективности процесса отслеживания ошибок в распределенных программных системах и линейках программных продуктов путем автоматизированной идентификации дублированных групп отчетов и групп отчетов, собранных с коррелированных ошибок, в сочетании с локализацией ошибок среди компонентов линеек программных продуктов. Сделан краткий обзор проблемы автоматизированного сбора и агрегации отчетов, рассмотрены несколько существующих программных средств для анализа отчетов, а также определены основные функциональные возможности типовой системы управления отчетами. Кроме того, предложена концепция корреляционной группы отчетов и приведен автоматизированный метод агрегации отчетов, основанный на правилах сравнения подписей отчётов, верхней формы отчётов, и трассировочных стеков отчетов о сбоях. Для оценки этих правил строятся две отдельные нечеткие модели - первая для расчета результата правила сравнения трассировочных стеков отчетов, а вторая - для интерпретации и сочетания результатов всех трех правил и создания интегрированного показателя степени сходства отчета о сбое с существующей корреляционной группой отчетов или иным отчетом. С помощью импорта групп отчетов с общедоступного репозитория Mozilla, тестируется разработанный прототип системы управления и агрегации отчетов. Во время эксперимента достигается точность в 90% и полнота в 81%. Наконец, предлагается подход к локализации крупнейших идентифицированных групп отчетов и представленных ими ошибок в линейке программных продуктов на основе информационной базы, состоящей из функциональной модели, списка программных компонентов и взаимосвязей между функциями и компонентами, делаются выводы и определяются цели для дальнейшей работы.

Ключевые слова: отчеты о сбоях, автоматизированный сбор и агрегация отчетов, локализация дефектов, нечеткая логика, распределенные программные системы, линейки программных продуктов, проект Mozilla Crash Stats, Socorro, система управления отчетами, отслеживание ошибок.

Introduction: Problem Actuality and Research Goal. Software defects (also known as bugs or issues) occur in any type of software development process, and, depending on the type, both the number of defects present in the system as well as the time it takes to detect them can vary greatly [1].

While most bugs can be discovered via in-house testing, there are always issues that will be detected only in the production environment by customers and actual users of the system. Since the amount of errors present in the system is never precisely known, it is impossible to accurately estimate the time and effort it will take to fix them. In extreme cases, tracking down and fixing bugs can take up as much as 80 % of project's financial costs [1]. In principle, it is impossible to guarantee that the program is 100 % correct. The general rule is that the larger and more complex a system is, the higher is the number of bugs.

Bug reports are crucial to identifying and solving problems in an efficient and robust manner. It has been observed that the more detailed the bug reports are, the faster developers will be able to fix the bug [2]. In many cases upon encountering a bug or a crash, users are expected to go to a dedicated forum and fill out a bug reporting form where they provide all necessary information and stack traces from their locally stored logs. Stack traces, system information, steps to reproduce the bug, user comments, and other information help developers reproduce and fix reported bugs by tracking their origin. However, not all users report bugs they've encountered, and for many it is simply too bothersome to manually provide the crucial technical information.

To ensure that developers get all necessary information needed to effectively fix bugs, many modern software products are shipped with embedded problem reporting tools. These tools automatically record and submit bug reports, with very little to no effort on the user's part. The most famous automated crash reporting system is the Windows Error Reporting system by Microsoft, which was found to be 5 times more efficient in helping fix bugs than reports submitted manually by a human [3].

However, embedded problem reporting tools are difficult and costly to implement, and the amount of data they collect can be challenging to analyze even for a small project. For example, Mozilla on average receives 96 million crash reports per month; they outnumber bug reports by more than 20,000: 1. Furthermore, 88.19 % of crash reports and 24.7 % of bug reports submitted to Mozilla are marked as duplicate [4].

Therefore, to make it easier to work with collected crash reports, most problem reporting tools aggregate the received reports according to a set of pre-defined rules. Many approaches to aggregation of crash reports were proposed, however, crash report aggregation in distributed software systems remains a non-trivial task. This is mainly because, depending on the usage scenario, a bug may cause multiple failures and crashes in several different modules at once, thus propagating its influence across the system and making the root cause of the defect harder to find.

A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [5]. In other words, an SPL is a family of related programs that are differentiated by a unique combination of features, which represent increments in functionality. There exist many modeling languages for SPL design, among which are FODA, FORM, FeatuRSEB, PLUSS, ODM and FAST [6].

In addition to the main issues inherent to testing and support of distributed software systems, SPLs present a different challenge, wherein a bug in each type of SPL components (core, configurable, and custom) needs to be handled in its own way within the concrete product and sometimes even the entire SPL. Furthermore, some reports collected during testing may be caused not by an actual bug in the code, but rather an incorrect configuration of the product (i.e. illegal combination of features) or errors inside the tests themselves. As such, this type of errors should be identified and dealt with separately.

In this paper, a brief overview of existing tools and solutions is given and, as no available product is found satisfactory, a method for automated crash report aggregation (ACRA) based on a combination of 3 simple rules and fuzzy logic is proposed. Furthermore, an approach to bug localization within an SPL is outlined. The objective of this combined approach is to help developers of distributed systems and SPLs to identify bugs as well as their scope and origin in a more efficient way.

Overview of existing approaches. Built-in crash reporting tools usually collect large amounts of data, which can be extremely helpful in identifying and localizing software defects. As the number of submitted reports grows, manual analysis of each bug report quickly becomes inefficient and in most cases downright impossible. However, even automated processing of large volumes of data gathered by popular software products oftentimes presents unique challenges and harsh demands on processing power, network bandwidth, and storage facilities. While high demand for network bandwidth is unlikely to ever be solved, many attempts have been made to mitigate and reduce the impact of the other two issues.

There exist several strategies that are commonly used to reduce the load associated with analysis of a large amount of crash reports, namely:

- Biased sampling – with so many reports it isn't always possible to process or display all of them. Companies like Mozilla only process a sample of 10 % out of their 96 million monthly crash reports [4]. This sample is randomized, but biased towards reports with user-provided details;
- Removal of duplicates – once a report has been identified as a duplicate of a previously submitted report, it is deleted and a special counter for the number of times the issue has been encountered is updated. Out of the 10 % sample Mozilla takes, using fuzzy matching techniques 88.19 % of reports [4] are classified as duplicate and are subsequently reduced;
- Report aggregation according to contained stack traces, failing method, or other technical information. This is usually the last step that is primarily aimed at the detection of the remaining correlated and duplicate reports.

The combination of these 3 methods helps save time and processing power for search, filter and other analytical

functions, as well as greatly reduces the required storage space. However, most of the existing solutions use relatively simple and easy to implement algorithms, and while such approach is fast and doesn't require a lot of processing power, much like Mozilla's solution it usually can't find all correlations present in the data [7].

There are built-in and standalone tools for log collection and analysis available for almost any programming language and platform, some focusing primarily on the collection and filtering aspect, while others are better at report analysis and dynamic aggregation. Fig. 1 summarizes the basic functionality required by a typical automated report management system. However, none of the tools we looked at implement the full list.

For example, Graylog [8], an enterprise solution for storing, accessing, and analyzing log data, allows for convenient log and report storage, visual representation, and search, but lacks in flexible aggregation. The tool has very limited ability to detect related logs, provided they aren't clear duplicates. For more advanced aggregation and clustering capabilities, users must install a plugin [9] and write their own explicit rules, resulting in a poorly generalized and constricted system with no ability to implement complex clustering or aggregation logic.

Sentry [10], a tool for collecting JavaScript user logs, exceeds in capturing an unprecedented level of information about JavaScript execution and its environment, but fails to provide a sufficiently customizable and flexible aggregation framework, forcing developers to implement workarounds in their code to integrate and expand its functions beyond the default feature set.

Both Android [11, 12] and iOS [13] platforms offer built-in tools for error and performance log collection, while their respective development environments and online platforms provide a convenient way of accessing all collected logs and viewing general statistics about the published applications. Furthermore, advanced functionality that is missing by default is oftentimes provided by third-party solutions [14, 15].

The only product that appears to be handling the issue of duplicate reports and correlated bugs in a configurable and robust way is Crashlytics [15]. The downside is that Crashlytics supports only mobile development (both Android and iOS), and, just like all other reviewed products, doesn't have the necessary tools to deal with SPL testing and support.

To summarize, while popular and efficient, the reviewed solutions for issue collection offer little in the way of intelligent report aggregation, most systems relying solely on stack trace analysis. In contrast to this, systems with complex aggregation and analysis capabilities are rare and usually require a complicated setup process and use of a separate tool for issue collection and even storage. Furthermore, none of the reviewed applications had the necessary tools and flexibility for dealing with SPLs.

The aim and benefits of automated defect identification. In most cases developers will prioritize bugs that occur frequently for a large percent of their user base. However, because a bug can lead to a variety of crashes under different usage scenarios, multiple report groups are sometimes related to the same bug, which makes the evaluation of bug and crash severity harder. We refer to a set of report groups related to the same bug as duplicate report groups. Furthermore, there are cases when an occurrence of one bug causes the other bug to occur. This is known as correlated bugs, and in such cases both bugs and every related to them report group needs to be analyzed, evaluated, and used together to fix the issue. We refer to all report groups pertaining to a bug as well as any report groups collected from its correlated bugs as a report correlation group (RCG). A schematic overview of an RCG is presented in Fig. 2.

Crash reports usually contain method signature, stack trace of the failing thread, crash time, information about runtime environment, and optionally user comments and attachments regarding the crash. These reports are aggregated into RCGs according to their similarity. The obtained groups are then ranked according to their report counts (i.e. frequency of crash occurrence) and bug item entries are created for the top RCGs.

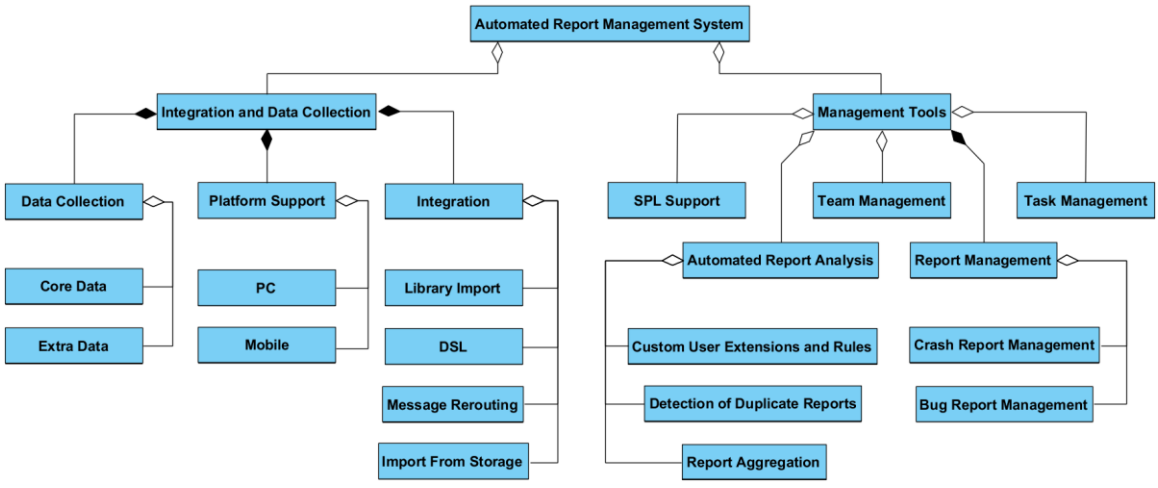

Fig. 1. Basic functionality of automated report management systems

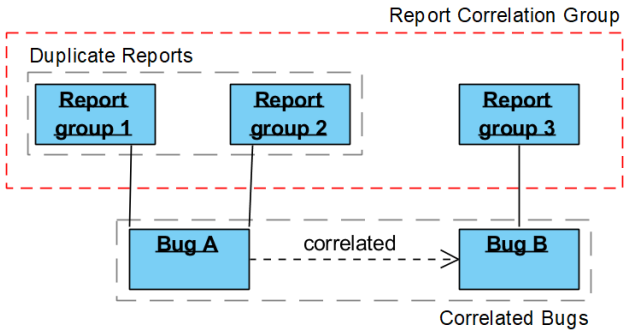

Fig. 2. Report correlation group (RCG)

Identification of RCGs (and therefore duplicate crash reports and correlated bugs) on the early stages of the debugging process can help developers fix existing corelated bugs together, as well as increase the overall speed and efficiency of the debugging process. It has been suggested that by analyzing and comparing a set of correlated bug reports, it becomes easier to track the root cause of the bug, especially in complex distributed systems and SPLs [2].

Identification of RCGs and, subsequently, bugs they represent can also help development teams to better manage their resources. One such example could be increasing the priority of correlated bugs or assigning more experienced developers to fix them. Furthermore, as crash reports are continuously submitted until the underlying bug that causes crashes is fixed, by quickly and efficiently eliminating the bug development teams can reduce the amount of crash reports they receive and, therefore, hardware and human demands associated with crash report storage and processing.

A fuzzy-based integrated approach to automated crash report aggregation (ACRA). To better identify duplicated and correlated crash reports, first we need to establish several definitions.

A stack trace is an ordered set of frames (F_i), each frame consisting of a method signature ($methSign$) and a fully qualified file name ($qFileName$). $F_i = methSign_i|qFileName_i$, where $i \in \{1 \dots n\}$ is the position of frame F_i in a stack trace of a total frame length n. Usually all reports belonging to a report group will have at least one identical frame. We refer to the top frame common to all stack traces of a report group as the top frame of the report group. The frames after it may be identical or vary across the group, but the method signature of the top frame is always used as part of the report group signature. An example stack trace is shown in Fig. 3.

Frame	Module	Signature	Source (Top Frame)
0	libmozglue.dylib	mozalloc_abort	memory/mozalloc/mozalloc_abort.cpp:34
1	libmozglue.dylib	abort	memory/mozalloc/mozalloc_abort.cpp:81
2	XUL	std::panicking::rust_panic	src/libpanic_abort/lib.rs:59

Fig. 3. Example of Stack Trace from Firefox

A report group signature S may be represented as $S = P_1|P_2| \dots |P_n$, where each element P_i in turn consists of $< File >< Operator >< Method >< Parameter > < Memory\ Location >$. In a report group signature, at least one P_i should not be NULL. P_i can't be formed using only the name of an operator, which depends on the programming language and signature composition. Furthermore, attributes like $File$, $Operator$, $Method$, and $Parameter$ can be NULL. For example, the Mozilla's report group signature for the above stack trace is "mozalloc_abort | abort | core::option::expect_failed".

A contains relation between signature elements P of a report group $S = P_1|P_2| \dots |P_n$ is defined as if $(file_i = file_j) \wedge \{op_i, meth_i, param_i\} \subseteq \{op_j, meth_j, param_j\}$, then P_j contains P_i.

Building on top of this, a binary relation $\subset$ on the set of all RCG signatures S is defined as $S_A \subset S_B$ if $\forall P_i^A, i \in \{1 \dots n\}, \exists j \in \{1 \dots m\} \mid P_j^B\ contains\ P_i^A$.

As the basis for the ACRA method we used 3 rules suggested earlier by Shaohua Wang et al. [19], namely:

1. Crash Signature Comparison. Given two report groups RG_A and RG_B with corresponding signatures S_A and S_B, RG_A and RG_B are correlated if $S_A \subset S_B$ or $S_B \subset S_A$. This is the least resource-demanding rule. It is aimed at determining correlations by investigating the similarity of report group signatures. An example of this rule is signatures "nsDiskCacheStreamIO::FlushBufferToFile()" and "Strstr|nsDiskCacheStreamIO::FlushBufferToFile", which differ only slightly and, therefore, are correlated.

2. Top Frame Comparison. Given two report groups RG_A and RG_B with top frames F_1^A and F_1^B, RG_A and RG_B are correlated if $\text{qFileName}_1^A = \text{qFileName}_1^B$. During comparison of fully qualified file names ($qFileName$), all file extensions are removed. This rule is aimed at a more detailed analysis of stack traces, in this case their source code paths. Much like the previous rule, it analyses and compares only the top frame of stack traces.

3. Frequent Closed Ordered Sub-Set Comparison. This rule is an extension of the previous rule, namely it is aimed at analyzing fully qualified file names of all frames, not just the top frame. A notion of relative support of the rule is introduced, wherein $relative\ support = number\ of\ shared\ frames / total\ number\ of\ frames$. Only the identical and longest set of frames with the relative support of greater or equal to 0.5 is called the frequent closed ordered sub-set of frames. Furthermore, the distance of the top frequent closed ordered frame to F_0 is also considered during the final evaluation of the rules.

Wang et al. [16] evaluated each of these rules separately, with report groups considered to be correlated if at least one of the 3 rules provided a positive result. In this paper, we propose a fuzzy-based integrated approach to automated crash report aggregation (ACRA). Namely, we use fuzzy logic to evaluate the combined output of these rules and compute the degree of report's similarity (i.e. membership) to other reports or report groups.

Furthermore, contrary to the approach to evaluation of the Frequent Closed Ordered Sub-Set Comparison rule employed by Wang et al. [16], we use a separate fuzzy model to calculate the degree of report's similarity based on the length of the common frame sequence as well as its distance to the top frame of the stack trace.

The ACRA method proposed in this paper relies on

the use of a fuzzy model to compute the final degree of report's membership to a given group, or alternatively the degree of correlation between two report groups. As its input, the model takes values received from the rules, and by applying weight coefficients and a threshold function produces the degree of report's membership.

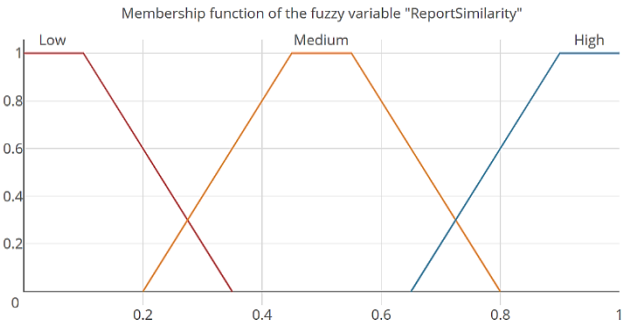

Fig. 4. Membership function of β = "ReportSimilarity"

A linguistic variable is a set of $\langle \beta, T, X, G, \mu \rangle$, where β is the name of the fuzzy variable, T is a set of terms (fuzzy values) of the variable, X is a fuzzy set that describes the so called "base values" of the term, G is a set of syntax rules that define the elements of T (it is possible for G to be empty, i.e. $G = 0$), and μ is the membership function that maps base values to the terms contained in T.

According to the above definition, the fuzzy variable $ReportSimilarity$, which represents the combined degree of the similarity between two reports or a report and a report group, is defined as $\beta =$ "$ReportSimilarity$", $T =<$ "low", "$medium$","$high$" $>$, $X = [0-1]$. The membership function μ_β is shown in Fig. 4.

Experimental results. To assess ACRA's performance in identification of duplicate and correlated crash reports, an experiment was performed. A prototype report management system was developed and tested on a pre-downloaded and verified data set. The data set for analysis was obtained from the Socorro server, which is maintained as part of the open to public Mozilla Crash Stats (MCS) project. A statistical comparison of the number of RCGs created by the MCS project and the prototype ACRA system was performed, and its results are presented in Table 1 and in Fig. 5 and 6. Fig. 5 shows the amount of reports ACRA and MCS correctly aggregated for the top 10 control groups (the closer to the control line, the better the result), while Fig. 6 compares their precision and recall.

Reports relating to the newest stable version of Mozilla Firefox were imported and analyzed.

A set of 426 reports separated by MCS into 34 RCGs has been selected, with the resulting set being biased towards duplicate groups that weren't correctly aggregated by MCS. To evaluate the performance of the ACRA method, the report groups provided by MCS were taken as an etalon (i.e. MCS precision = 100 %), however their relationships were analyzed and duplicate groups were found via analysis of the related Bugzilla bug IDs provided by Mozilla developers for any sufficiently large crash report group. More specifically, 21 etalon RCGs were identified and used as a control sample for the experiment.

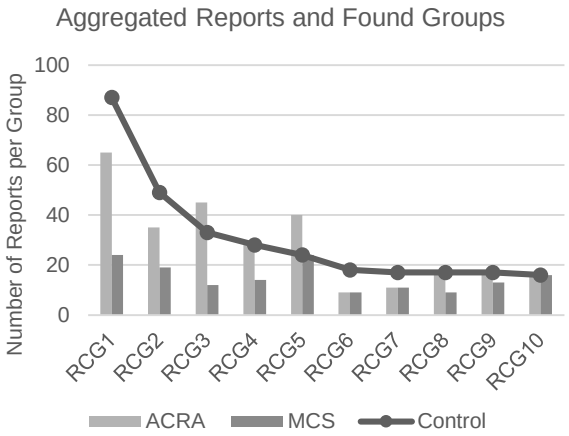

Fig. 5. Aggregated reports for the top 10 correct RCGs

Analysis of the imported from MCS report groups showed that the top 21 groups contained 267 aggregated reports, which is 62 % of all reports in the data set. Contrary to this, the top 21 groups created by ACRA contained 383 reports, albeit 38 of them (10 %) were assigned erroneously, which results in 345 correctly aggregated reports, or 81 % of the total number. In general, compared to the 21 etalon groups, 34 groups were created by Mozilla, whereas ACRA created only 25 groups, a result achieved by a more thorough and in-depth analysis of the structure and contents of crash reports.

Table 1 – Results of the experiment

	MCS	ACRA
Aggregated correctly	267 reports, 62 %	345 reports, 81 %
Created RCGs	34/21	25/21
Deleted reports	N/A	83 reports, 18 %

Using the metrics of information retrieval, precision and recall values were calculated in the following way:

$$precision = \frac{|\{correctRCGs\} \cap \{retrievedRCGs\}|}{|\{retrievedRCGs\}|},$$

$$recall = \frac{|\{correctRCGs\} \cap \{retrievedRCGs\}|}{|\{correctRCGs\}|}.$$

Precision reflects the fraction of created report groups that are correct, while recall indicates the overall fraction of correct report groups that were found.

Using crash reports obtained from the Socorro server, a precision rate of 90 % and a recall of 81 % were achieved. The obtained recall value is significantly higher than the 62 % recall of the MCS project.

A statistical analysis of the duplicate report removal feature (reports with similarity of above 0.95 %) showed that 18 % of the previously thought to be unique reports could be reduced to free up the storage space of the

system, a noticeable improvement for enterprise-grade products.

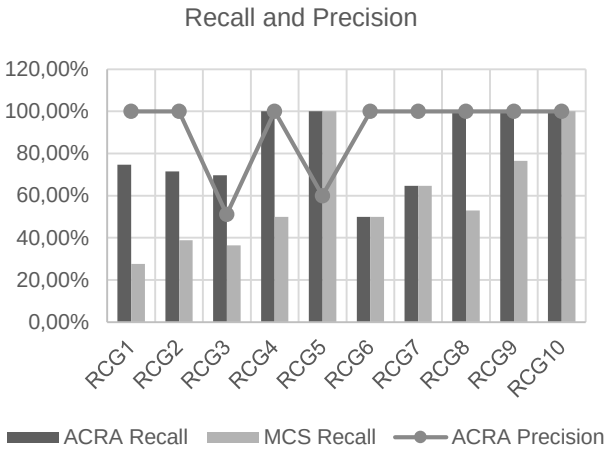

Fig. 6. Recall and precision for the top 10 correct RCGs

The identification of RCGs allows developers to access the groups of related crash reports and information contained within them. Working with RCGs instead of separate crash reports can help identify and track the underlying cause of the crash faster, thus decreasing the time it takes to debug and fix an error [2]. Furthermore, it has been observed that identification of RCGs can significantly increase the overall efficiency of the work associated with debugging of certain types of bugs [4].

Bug localization approach within an SPL. Feature models play a key role in testing of SPLs as they constrain the space of products to test and enable accurate categorization of failing tests as failures of programs or the tests themselves, not as failures due to illegal configurations [5]. Consequently, testing of SPLs while ignoring such dependencies is senseless.

As part of our future work, to enable traceability between collected crash reports and features (software components) of an SPL, we propose the following operating model (OM) [17] for crash report localization (CRL). The CRL OM can be represented as a tuple:

$$OM(CRL) = \langle InfBasis, ProcAlgorithm, Metrics \rangle.$$

$InfBasis$ should be built separately for each SPL product. It consists from a valid feature model of the SPL, a list of software artifacts (product components) belonging to each SPL product, a mapping between components and features, and a list of unprocessed crash reports. $ProcAlgorithm$ represents an expanded ACRA fuzzy logic method that is responsible for using the technical information present inside crash reports (mainly stack traces and method signatures) to aggregate reports into RCGs and establish to what type of SPL components these groups belong. $Metrics$ measure and provide statistics on types of failures and their counts for various SPL components and features, as well as reflect how reliable the work of $ProcAlgorithm$ is.

With the help of this OM, it is possible to connect crash reports and implemented SPL components (features) by using the $ProcAlgorithm$ and data from the $InfBasis$ to trace sets of crash reports in relation to the implemented software components of an SPL.

In other words, given a valid feature model, a set of existing software artifacts, and mappings between software components and features, it is possible to use the technical information contained inside crash reports to identify the type and name of the component that is most likely responsible for the crash described by an RCG.

Conclusions and future work. Based on the results of preliminary tests of the developed prototype ACRA system, the proposed rules for identification of duplicate and correlated crash reports combined with the fuzzy evaluation approach have shown better results than the current solution employed by Mozilla.

By optimizing the underlying fuzzy models and implementing the suggested SPL bug localization technique, a more efficient approach to improvement of the bug tracking process of both distributed systems and SPLs will be obtained. Furthermore, localization of the RCGs identified by the ACRA method in an SPL will allow developers to determine the scope of crashes and corresponding to them bugs in a more efficient manner.

Our future work concerns further testing and improvements of the ACRA method combined with development and testing of a system that will collect crash reports, aggregate them into crash groups using the ACRA method, and (if needed) localize the obtained RCGs within an SPL by using the suggested SPL bug localization approach. Furthermore, we also plan to expand the ACRA method to work with user-submitted bug reports.

References

1. Sommerville I. Software engineering / Ian Sommerville. – 9th ed. Addison Wesley, 2011. 773 p.
2. Adrian S., Nicolas B., Rahul P. Do stack Traces Help Developers Fix Bugs? *MSR 2010: 7th IEEE Working Conference on Mining Software Repositories*, 2010. P. 118–121.
3. Kinshumann K., Glerum K., Greenberg S., Aul G. et al. Debugging in the (very) large: ten years of implementation and experience. *ACM Communications*. New York City, NY, USA, 2011. № 7, vol. 54. P. 111–116.
4. Iftekhar A., Nitin M., Carlos J. The Impact of Automatic Crash Reports on Bug Triaging and Development in Mozilla. *The 14th International Symposium on Open Collaboration*. Berlin, 2014.
5. *Carnegie Mellon University Software Engineering Institute*. Software Product Lines. URL: https://www.sei.cmu.edu/productlines (accessed 11.05.2018).
6. Asmaa A., Ounsa R., Nissrine S., Camille S. Selecting SPL Modeling Languages: a Practical Guide. *The Third World Conference on Complex Systems (WCCS), 2016*. Marrakech, Morocco, November 2015.
7. Tejinder D., Foutse K., Ying Z. Classifying Field Crash Reports for Fixing Bugs: A Case Study of Mozilla Firefox. *The 27th IEEE International Conference on Software Maintenance (ICSM)*. Williamsburg, VA, USA, September 2011.
8. *Graylog*. URL: https://www.graylog.org/overview (accessed 11.05.2018).
9. *Aggregates Plugin for Graylog*. URL: https://marketplace.graylog.org/addons/0d01a899-138a-4f77-a9e7-04be4cc5e190 (accessed 11.05.2018).
10. *Sentry*. URL: https://sentry.io/for/javascript (accessed 11.05.2018).
11. *Google Play Console*. URL: https://play.google.com/apps/publish/ (accessed 11.05.2018).
12. *Firebase crash reporting*. URL: https://firebase.google.com/docs/crash/ (accessed 11.05.2018).
13. *Xcode*. URL: https://developer.apple.com/xcode/ (accessed 11.05.2018).

14. *Firebase Crashlytics*. URL: https://firebase.google.com/docs/crashlytics/ (accessed 11.05.2018).
15. *Crashlytics reports*. URL: http://try.crashlytics.com/reports/ (accessed 11.05.2018).
16. Wang S., Khomh F., Zou Y. Improving bug localization using correlations in crash reports. *The 10th IEEE Working Conference on Mining Software Repositories*. San Francisco, CA, USA, May 2013. P. 247–256.
17. Tkachuk M.V., Abbasov T.F. An operating model for dynamic requirements management in agile software development. *The XXV International Scientific and Practical Conference on Information Technologies MicroCAD-2018*. Kharkiv, May 2018. P.12.

References (transliterated)

1. Sommerville I. Software engineering / Ian Sommerville. – 9th ed. Addison Wesley, 2011. 773 p.
2. Adrian S., Nicolas B., Rahul P. Do stack Traces Help Developers Fix Bugs? *MSR 2010: 7th IEEE Working Conference on Mining Software Repositories*, 2010, pp. 118–121.
3. Kinshumann K., Glerum K., Greenberg S., Aul G. et al. Debugging in the (very) large: ten years of implementation and experience. *ACM Communications*. New York City, NY, USA, 2011. № 7, vol. 54, pp. 111–116.
4. Iftekhar A., Nitin M., Carlos J. The Impact of Automatic Crash Reports on Bug Triaging and Development in Mozilla. *The 14th International Symposium on Open Collaboration*. Berlin, 2014.
5. *Carnegie Mellon University Software Engineering Institute*. Software Product Lines. Available at: https://www.sei.cmu.edu/productlines (accessed 11.05.2018).
6. Asmaa A., Ounsa R., Nissrine S., Camille S. Selecting SPL Modeling Languages: a Practical Guide. *The Third World Conference on Complex Systems (WCCS), 2016*. Marrakech, Morocco, November 2015.
7. Tejinder D., Foutse K., Ying Z. Classifying Field Crash Reports for Fixing Bugs: A Case Study of Mozilla Firefox. *The 27th IEEE International Conference on Software Maintenance (ICSM)*. Williamsburg, VA, USA, September 2011.
8. *Graylog*. Available at: https://www.graylog.org/overview (accessed 11.05.2018).
9. *Aggregates Plugin for Graylog*. Available at: https://marketplace.graylog.org/addons/0d01a899-138a-4f77-a9e7-04be4cc5e190 (accessed 11.05.2018).
10. *Sentry*. Available at: https://sentry.io/for/javascript (accessed 11.05.2018).
11. *Google Play Console*. Available at: https://play.google.com/apps/publish/ (accessed 11.05.2018).
12. *Firebase crash reporting*. Available at: https://firebase.google.com/docs/crash/ (accessed 11.05.2018).
13. *Xcode*. Available at: https://developer.apple.com/xcode/ (accessed 11.05.2018).
14. *Firebase Crashlytics*. Available at: https://firebase.google.com/docs/crashlytics/ (accessed 11.05.2018).
15. *Crashlytics reports*. Available at: http://try.crashlytics.com/reports/ (accessed 11.05.2018).
16. Wang S., Khomh F., Zou Y. Improving bug localization using correlations in crash reports. *The 10th IEEE Working Conference on Mining Software Repositories*. San Francisco, CA, USA, May 2013, pp. 247–256.
17. Tkachuk M.V., Abbasov T.F. An operating model for dynamic requirements management in agile software development. *The XXV International Scientific and Practical Conference on Information Technologies MicroCAD-2018*. Kharkiv, May 2018, p.12.

Received 19.05.2018

Відомості про авторів / Сведения об авторах/ About the Authors

Зіньковський Олексій Ілліч (***Зиньковский Алексей Ильич***, ***Zinkovskyi Oleksii Illich***) – Національний технічний університет «Харківський політехнічний інститут», студент; м. Харків, Україна; ORCID: https://orcid.org/0000-0002-6234-8817; e-mail: aleksey.zinkovskiy95@gmail.com

Гамзаєв Рустам Олександрович (***Гамзаев Рустам Александрович***, ***Gamzayev Rustam Olexandrovich***) – кандидат технічних наук, доцент, Національний технічний університет «Харківський політехнічний інститут», доцент кафедри програмної інженерії та інформаційних технологій управління; м. Харків, Україна; ORCID: https://orcid.org/0000-0002-2713-5664; e-mail: rustam.gamzaev@gmail.com

Андреас Боллін (***Андреас Боллин***, ***Andreas Bollin***) – доктор технічних наук, професор, Альпен-Адріа університет, керівник Інституту дидактики інформатики (Alpen-Adria University, Head of Department of Informatics Didactics); м. Клагенфурт, Австрія (Klagenfurt, Austria); ORCID: https://orcid.org/0000-0003-4031-5982; e-mail: andreas.bollin@aau.at

Ткачук Микола Вячеславович (***Ткачук Николай Вячеславович***, ***Tkachuk Mykola Vyacheslavovich***) – доктор технічних наук, професор, Національний технічний університет «Харківський політехнічний інститут», професор кафедри програмної інженерії та інформаційних технологій управління; Харківський національний університет імені В.Н. Каразіна, професор кафедри моделювання систем і технологій, м. Харків, Україна; ORCID: https://orcid.org/0000-0003-0852-1081; e-mail: tka.mobile@gmail.com