STUDIA INFORMATICA 2014
Volume 35 Number 1 (115)

Michał KOMOROWSKI
Politechnika Warszawska, Instytut Informatyki

APPLYING RECURRENCE PLOTS IN ANALYSIS OF COMPUTER PROGRAMS

Summary. Recurrence plots are a tool which allow one to analyze and visualize recurrence in non-linear dynamical systems. In this paper it was examined whether recurrence plots could be used to analyze data collected by historic debuggers. Besides a tool that facilitates this kind of analysis was proposed.

Keywords: historic debuggers, IntelliTrace, recurrence plots

ZASTOSOWANIE WYKRESÓW REKURENCYJNYCH W ANALIZIE PROGRAMÓW KOMPUTEROWYCH

Streszczenie. Wykresy rekurencyjne to narzędzie pozwalające wizualizować oraz analizować powtarzające się stany w nieliniowych systemach dynamicznych. Artykuł ten opisuje eksperymenty, w których sprawdzono, czy wykresy rekurencyjne nadają się do analizy programów komputerowych. Zaproponowano również narzędzie wspomagające taką analizę.

Słowa kluczowe: debugery historyczne, IntelliTrace, wykresy rekurencyjne

1. Introduction

A set of variables that describes a dynamical system at some point of time is called a state of a system. If a particular system returns (recurs) to already visited states, it means that it has a property called recurrence. Recurrence plots [15, 17] (RP), that were described for the first time in 80's [17], are a tool which allows one to analyse and visualise recurrence in non-linear dynamical systems, in order to better understand them, based on one dimensional signals (time series).

RP are widely known and used in many domains of science, for example in [16] neuro-science, geology, palaeo-climatology, analysis of financial data and many more. There are also examples of application of RP in Information Technology field. Article [4] describes how recurrence plots could be used for data classifications. Another example is analysis of activity of computers users [6] or of network traffic [20]. The method described in [19] allows one to perform attack on TCP protocol. This method is not based on RP but the author adapts the same technique as is used to generate recurrence plots - delayed coordinate embedding. [3] also does not use RP but shares the same idea of the graphical representation and analysis of complex data, in this case DNA sequences.

This article describes a new approach in which recurrence plots are used to analyse data collected by IntelliTrace [11, 14] historic debugger in order to support testing, detect bugs or repeating users actions. RP seem to be well-suited to this scenario because they are dedicated to non-linear dynamical systems and computer programs belong to this category [6].

RP could be used when we do not know all variables that affect a state of a system, there is too many of them or it is difficult to measure them. In the case of computer programs the number of variables is very big and at the same time the activity of users is difficult to take into account. What is also important in the case of data collected by historic debugger many different signals, required to create a recurrence plot, could be generated without need to modify a source code.

At the beginning of this paper, in section 2, recurrence plots are described. Then, in section 3 historic debuggers are introduced. The description of developed tools can be found in section 4. Finally, the experiments which show that RP could detect recurring states in IntelliTrace logs, are vulnerable to users activity etc. are described in section 5. The last section contains summary.

2. Recurrence plots

Recurrence plots is a method of analysis of non-linear data, the result of which is a two dimensional plot showing when states of a system being analysed were similar to each other. The input data for recurrence plots analysis are in the form of a one dimensional signal (time series). It means that we do not have to know all variables that affect the state of a system or know how to measure all these variables. Everything that is required is one representative variable, based on which reasoning about whole system is possible. A good example of this kind of a variable is a price of stocks on a stock market.

A generation of a recurrence plot begins with transforming input signal into multi-dimensional one. This process is known as delayed coordinate embedding. If a signal looks as follows:

$$X1, X2...Xn$$

then multi-dimensional vector *Yi* will look in the following way:

$$Yi = \{Xi, Xi - d, Xi - 2d..., Xi - (m-1)d\}$$

A vector *Yi* can be seen as a vector describing the state of a system in a original phase space, at some point *i*, using *m* variables. A parameter *m* is called embedding dimension and a parameter *d* - time delay. Let's consider this signal:

$$10,2,3,5,6,8,9...$$

If *m*=3 and *d*=1, then a multi-dimensional signal will have a form:

$$Y3 = \{3,2,10\}, Y4 = \{5,3,2\}, Y5 = \{6,5,3\}...$$

Vectors Y1 and Y2 were omitted, because for given values of parameters *m* and *d* it is not possible to create them. The correct selection of *m* and *d* is important, because otherwise analysis of RP would give wrong results. In order to calculate an optimal value of the delay parameter the algorithm using mutual information could be used [13]. Article [9] describes approach to determine parameter *m* based on the K-nearest neighbours algorithm.

The next step is to calculate a distance *Di* (a measure of similarity) between every pair (Yi, Yj) of reconstructed vectors. A distance is usually calculated based on [16] Manhattan norm, Euclidean Norm or Maximum norm. If the distance between two vectors (states) is lower or equal than a given threshold *Tr* , a point (i, j) on a plot is marked with a colour. For example a point 120, 250 on a plot would correspond to a distance between vectors 120 and 250 in the reconstructed multi-dimensional space. There are also some rules [16] which helps in determining a value of the threshold *Tr* e.g. it should not exceed 10% of the maximum phase space diameter.

There are some extensions to standard recurrence plots. For example Cross Recurrence Plots (CRP) and Join Recurrence Plots (JRP) [15, 16] allow one to analyse dependencies between two systems. CRP method requires that time series being analysed come from the same system or systems that are very similar, while JRP could be used when both systems are described by time series of different nature e.g. values of signals are expressed in different units.

In order to quantify RP, some measures describing them were proposed. This approach is known as Recurrence Quantification Analysis (RQA) [15, 16] and is based on density of recurring points and diagonal, horizontal or vertical structures appearing on RP.

In the basic approach generated plots are sets of black and white points. There are also tools [21] which create multi-colour recurrence plots. In this case a colour of a point depends on a distance between vectors (states).

3. Historic debuggers

Historic debuggers were described for the first time in 60's [12]. They allow one to move back and forward in a history of a program execution. Thanks to that it is possible to quickly fix bugs without restarting an application many times or reproduce rare problems.

Historic debuggers could be divided into two main categories. These in the first category, log invocations of methods and changes to program states during a program execution [1, 2, 14, 18]. In the second approach a reverse version of a program code [10] is generated. Some historic debuggers, in order to record a program execution, modify a program code before executing it [14, 18], while others use virtual machines [1, 2].

IntelliTrace [14] is a historic debugger which is a part of Visual Studio Ultimate Edition and is dedicated to .NET platform. It uses logging approach together with the instrumentation of a program code. IntelliTrace attaches to an application and modifies its Common Intermediate Language (CIL) by injecting special instructions. These instructions are responsible for recording data. Collected data are stored in *iTrace* files which could be examined in Visual Studio or programmatically. It works in two modes: basic and extended, that differ in performance and range of information that is collected. In the article [11] IntelliTrace was described in much more details.

The listing below shows a fragment of a computer program that was monitored with IntelliTrace. The method *StartCalculations* is responsible for execution of some algorithm. Figure 1, on the left side, shows how Visual Studio displays this code when being debugged with IntellITrace. On the right side we can see a corresponding fragment of IntelliTrace log. Table 1 shows the content of the recorded log when loaded into the database but only the basic columns where shown. The first two columns contain identifiers of the call and of the parent call. The next two columns show respectively the name of the callee and caller.

```
public Result StartCalculations()
{
   IAlgorithm alg;

   if (Mode == Mode.Simple)
      alg = new SimpleAlgorithm();
   else
      alg = new ComplexAlgorithm();

   return alg.Run();
}
```

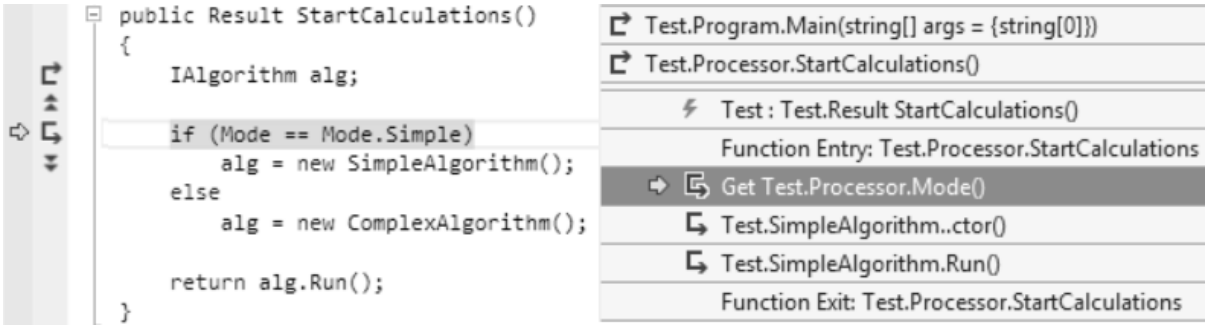

Fig. 1. A fragment of a code and of IntelliTrace log shown in Visual Studio
Rys. 1. Fragment kodu oraz logu IntelliTrace wyświetlony w Visual Studio

Table 1

Content of a log when loaded into a database

Id	Parent Id	Method	Parent Method
1	NULL	Test.Program.Main	NULL
2	1	Test.Processor..ctor	Test.Program.Main
3	2	Test.Processor.Mode	Test.Processor..ctor
4	1	Test.Processor.StartCalculations	Test.Program.Main
5	4	Test.Processor.Mode	Test.Processor.StartCalculations
6	4	Test.SimpleAlgorithm..ctor	Test.Processor.StartCalculations
7	4	Test.SimpleAlgorithm.Run	Test.Processor.StartCalculations

4. Tools

There are many tools [5, 7, 21] which allow one to generate recurrence plots. Some of them [21] were used in experiments described later in section 5, however it was decided to develop a new tool because existing solutions were missing some important functionalities. The developed component is called *RecurrencePlot.dll* and was used to extend the previously developed application *ExecutionTraceToolkit* [11].

ExecutionTraceToolkit is a tool which simplifies and makes working with IntelliTrace more efficient. It also allows users to load *iTrace* logs into a database or generate call trees. This tools was extended and used in experiments that are described in the latter part of this article. The improved version of *ExecutionTraceToolkit* allows users to:

- Generate RP for signals consisting of hundreds of thousands of elements. It was achieved by using sampling.
- Generate signals (time series) for data collected by IntelliTrace historic debugger. There are many possibilities how this kind of signal could be generated based on these data. However, in this article one particular time series, consisting of methods identifiers, was

examined. It was chosen because executions of methods determine what happened in the computer program.

- Define a specific algorithm which will be used to calculate distance between vectors (state) for a given signal. This function should be used when standard algorithms e.g. Euclidean distance are not suitable for data being considered. For example, in the experiments (see section 5) a custom algorithm for a signal consisting of method identifiers was used. This algorithm starts with a value equal to the parameter *m*. Then it iterates through vectors (v1, v2) and decreases a value by one if vectors are the same on a given position. If vectors differ on some position algorithm stops and returns a result. The intuition is that two sequence of methods calls are similar if they consist of the same methods. If at some position different methods were executed, it means that sequences represent a different business logic.
- Generate a recurrence plot for a selected part of a signal. It is useful when we do not want to analyse a whole signal.
- Generate a recurrence plot for a multi-dimensional signal. In this case a delayed coordinate embedding step of the processing is skipped (see section 2). This functionality is useful when we want to apply recurrence plot analysis to some existing multi-dimensional data.
- Provide descriptions for elements of a given signal. These descriptions are then visible on a plot. It makes analysis easier.
- Point out a point on a plot in order to see a pair of vectors based on which the point was created. It also makes analysis easier.
- Export reconstructed phase space (a set of multi-dimensional vectors) to a file.
- Zoom in or zoom out any part of a plot. If a signal is long, then a plot may be too general. In this case a user may want to zoom in and examine a selected part of a plot.
- Export a plot to a file.
- Calculate a phase space diameter in order to estimate a correct value of the threshold *Tr*.

Some of mentioned functionalities are also available in other tools but not all of them. What is also important *ExecutionTraceToolkit* aggregates many functions in one place what makes analysis more comfortable.

5. Experiments

In the experiments described later in this section three different programs were used. The first one is a simple application *Fibonacci* which calculates Fibonacci numbers in a recursive

way. It was used to perform a benchmark and check if RP could be used to analysed computer programs at the example of the program with only a few methods and simple logic.

The next one is a program developed by the author, called *LanguageTrainer*. It is used to maintain a set of words in foreign languages and help users in repeating and learning of these words. The last is a commercial financial software called *Rafaello* (the name was changed), which is a tool used by banks to maintain their business.

In order to calculate optimal value of parameters *m* and *d* the program VRA (*Visual Recurrence Analysis*) was used. In order to calculate these parameters VRA uses algorithms mentioned in the section 2 i.e. K-nearest neighbours and algorithm based on mutual information. All recurrence plots shown in the following sections were generated by *ExecutionTraceToolkit*.

The goal of conducted experiments was to check if RP could detect patterns in signals generated based on logs of historic debuggers, how activity of a user affects recurrence plots and if RP can be used to detect errors in the business logic of computer programs.

5.1. Fibonacci

In this experiment *Fibonacci* application was used. It is a simple and a recursive program so RP should show repeating patterns. Initially, the program was run under control of IntelliTrace and two Fibonacci numbers were calculated. Then *iTrace* log was processed and a signal consisting of 4527 items was generated.

Figure 1 shows a plot that was created based on this signal (m=2, d=100 and Tr=0). On the left side of this figure we can see a general plot and on the right side a zoomed in fragment.

A plot is visibly divided into a few parts. It could be easily explained. A scenario used to record *iTrace* log consisted of two steps. In each step one Fibonacci number was calculated. Parts of a plot corresponds to these two steps. There are four of them (not two) because RP is symmetric.

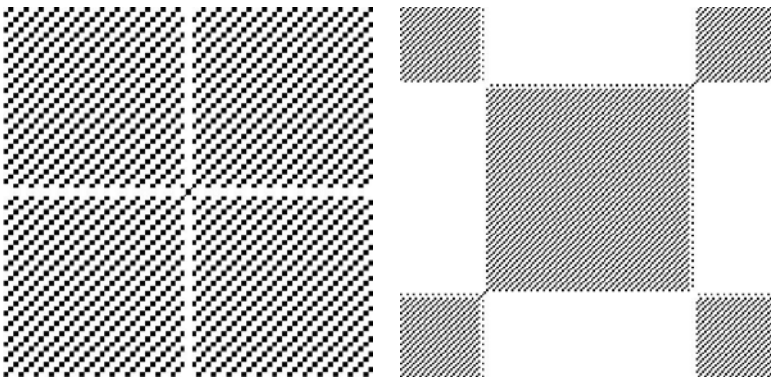

Fig. 2. A plot and a zoomed in fragment of it for the signal for Fibonacci program
Rys. 2. Wykres oraz jego przybliżenia dla sygnału dla programu Fibonacci

A zoomed in fragment of a plot shows clear diagonal structures. It means that a program was executing the same path many times what is consistent with the implementation. It is also worth noticing that the generated plot is not completely black. It means that except a recursive method some other methods were executed. This observation is also consistent with the implementation.

It is interesting to compare this plot with plots produced for a sine signal (m=2, d=4 and Tr=0.28) and for a white noise signal (m=1, d=1 and Tr=1.25) (see figure 2). RP for a sine is extremely well structured. A plot for a white noise is the opposite. Plots for time series created based on *iTrace* logs are ”somewhere” between.

As to the plot for the white noise signal it is also important that it was generated with a very high value of the threshold Tr i.e. equal to 50% of a phase space diameter. For smaller values a recurrence plot was empty or contained a small number of recurring points. For all other cases Tr was not exceeding recommended 10% of a phase space diameter [16].

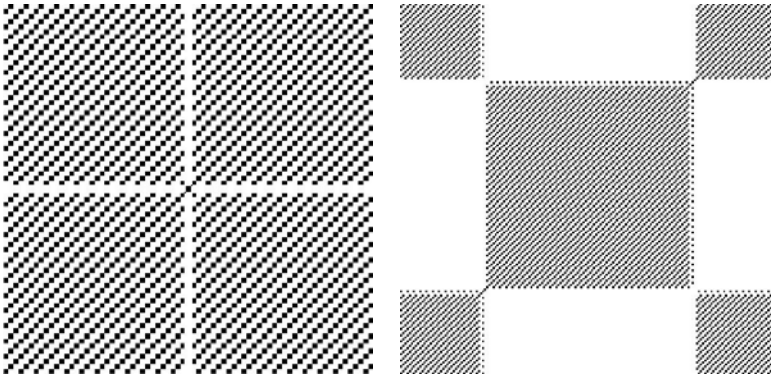

Fig. 3. A plot and a zoomed in fragment of it for the signal for Fibonacci program
Rys. 3. Wykres oraz jego przybliżenia dla sygnału dla programu Fibonacci

This experiment also shows that it is important to limit the length of time series and/or to propose automatic way of analyzing RP. The signal for the program is not very long but even in this case a generated plot has more than 16 million of points (4527×4527). The signal could be shortened by either using sampling, or by carefully selecting methods to be monitored by a historic debugger, so that a recorded log would be smaller.

Finally, an error was injected into *Fibonacci* application i.e. the program has a property *NoOfCalls* which stores number of recurrence calls, in the version with an error this property was incremented more times than it was required. Figure 3 shows fragments of plots generated for both versions of the program (m=2, d=100 and Tr=0) that were found to be different.

Further investigation showed that in the signal for the program without an error there were exactly 3 recurring states/vectors. In the second signal, 3 additional states appeared and this disrupted a pattern. It leads to the conclusion that the sequence of called methods was

somehow changed. However, it was difficult to determine where a bug was located without investigating a code of a program.

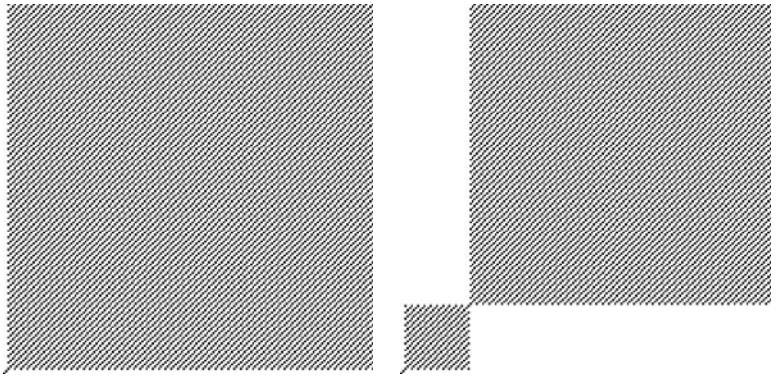

Fig. 4. Plots for a program without an error (on the left) and with an error (on the right)
Rys. 4. Wykres dla programu bez błędu (z lewej) i z błędem (z prawej)

5.2. LanguageTrainer

This sections contains a description of four different experiments that were performed with *LanguageTrainer* application.

5.2.1. Patterns detection

At the beginning, the paging component (at particular point of time only a subset of all words is displayed and a user is able to switch from a page to a page) of *LanguageTrainer* application was investigated.

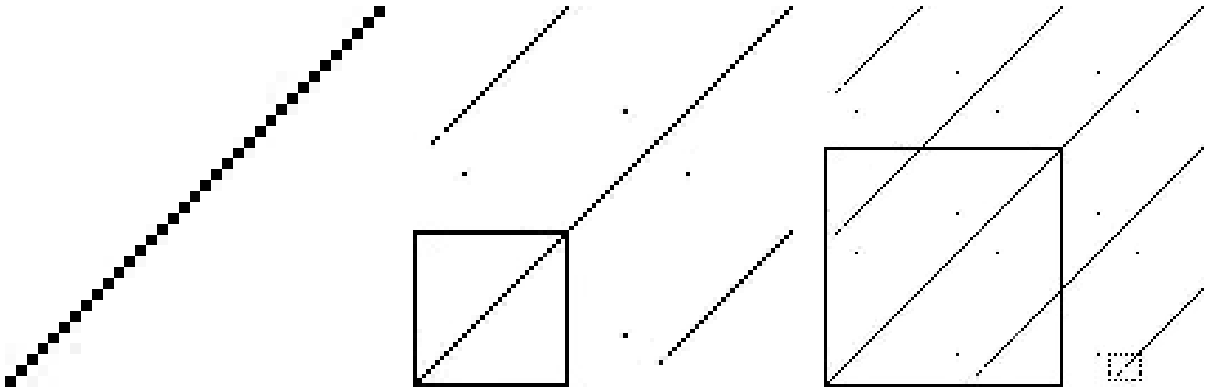

Fig. 5. Plots for the paging component
Rys. 5. Wykresy dla komponentu odpowiedzialnego za stronicowanie

The application was run under control of IntelliTrace, but only methods responsible for paging were monitored. The control parameters were m=9, d=3 and Tr=0. The application was executed three times. Each time a user was working with the program longer performing the same set of actions i.e. move one page forward, move 5 pages forwards, move to the last page, move to the fist page. Finally, for each scenario a recurrence plot was generated.

In the first RP no recurring states were detected for given parameters except the main diagonal line which appears on every recurrence plot.

The second and the third plots contains repeating structures. Moreover, the second plot contains in itself the first plot and analogically the third plot contains in itself the second plot (see solid rectangles). It suggests than RP are additive. It also means that RP detected repeating actions performed by a user.

It is also worth investigating more carefully a fragment in the small dotted rectangle. We can see a break in the long diagonal line (it is also visible in the second plot). This break appears because at the beginning *LanguageTrainer* application counts the number of all available expressions. This is a single actions which does not repeat/recur.

Interesting are also a single black points visible in the first and in the second plot. It means that some states are rare and/or persist only for a very short time.

5.2.2. Vulnerability to changes in a signal

The next experiment was similar to the one described in section 5.2.1, but this time the scenario was changed a little bit i.e. in the third step one additional action (move one page forward) was performed by a user. Figure 5 shows a recurrence plot (m=9, d=3 and Tr=0) generated for this scenario.

The plot differs from the one generated in the previous experiment. The diagonal lines visible in the old plot are now ”broken” into two shifted parts (see dotted rectangle). Besides the new short diagonal line (see solid rectangle) is now visible. It corresponds to the one extra action performed by a user. This experiment shows that RP are vulnerable to even relatively small changes in time series caused by a user activity.

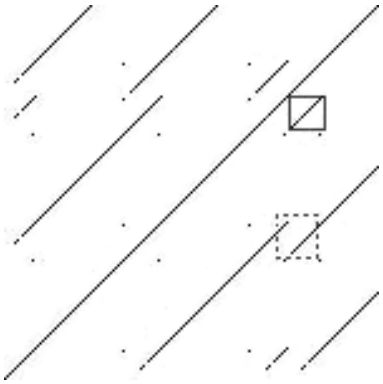

Fig. 6. A plot for the paging component for a changed scenario
Rys. 6. Wykres dla komponentu odpowiedzialnego za stronicowanie w zmienionym scenariuszu

5.2.3. Additivity of RP

The goal of this experiment was to confirm that RP are additive i.e. a plot generated for a given signal A (set of actions performed by a user) will be included in a plot generated for an extended signal C which has a form (B denotes additional actions performed by a user):

C = AB

Initially, a basic scenario consisting of five steps was prepared. Then *LanguageTrainer* was run under control of IntelliTrace a few times but each time a scenario was different i.e.:

- Session 1 - Step 1
- Session 2 - Step 1, 2
- Session 3 - Step 1, 2, 3
- Session 4 - Step 1, 2, 3, 4
- Session 5 - Step 1, 2, 3, 4, 5
- Session 6 - Step 5, 1, 3, 4, 2

Finally, for each session a recurrence plot was generated using the same values of parameters *m* and *d*. The results were consistent with observations from previous experiments i.e.: RP for the session 2 contained the plot for the session 1, the plot for session 3 contained the plot for the session 2 etc. What is also interesting a plot for the session 6 contained fragments of other plots, but these fragments appeared in the order of steps in the session 6.

5.2.4. Errors detection

In this experiment it was examined if RP could be used to detect errors in the business logic of a program. In order to do so, a few versions of *LanguageTrainer* were prepared. Each version had a bug in some component of the application. Then *LanguageTrainer* was run under control of IntelliTrace and a user was trying to perform set of actions (each time the same).

This time IntelliTrace was monitoring all methods because bugs were located in different components and it was difficult to choose a subset of methods to monitor. Finally, recurrence plots were generated (m=3, d=2 and Tr=0) and compared with a plot for the application without bugs.

The experiment showed a few problems with application of RP in analysis of computer programs. Produced signals were very long (hundreds of thousands of items). It means that in order to generate plots signals had to be sampled. Unfortunately, it turned out that sampling affects generated plots seriously which makes comparison very difficult or even impossible. Besides, even a small bug caused a considerable change in a plot and it was difficult to translate/map changes in a plot to particular bugs.

5.3. Rafaello

In the last experiment *Rafaello* application was used. RP were used to check if a new component in *Rafaello*, responsible for maintaining customers data, works correctly. The new

component replaced the old one, created in the old difficult to maintain technology, but the interface of communication with the rest of the system was untouched.

The idea was to compare RP generated for the application with the old component and with the new component. In order to collect required data two versions of the application were monitored by IntelliTrace while a user was: creating a new private client, creating a new corporate client or updating a private client. For both version of the program the common interface should have been used in the same way, so signals were produced based on methods of this interface (in the collected data 180 different methods were identified) i.e. RP were generated not for the whole system but for a part of it.

Figure 6 (m=2, d=5 and Tr=0) shows on the left side a plot for the new component and on the right side a plot for the old component. They are the same, except differences in the central part of plots. Further investigation of a signals showed that these differences were caused by a very minor differences between input signals that could be neglected. It confirms earlier observations that RP are vulnerable to even the smallest changes in the input signal.

The positive results of this experiment were consistent with the results of other kind of tests e.g. functional testing performed by a test team, automatic integration tests. It suggests that RP could be used as a first level test, which could be run easily and quickly, before other testing methods.

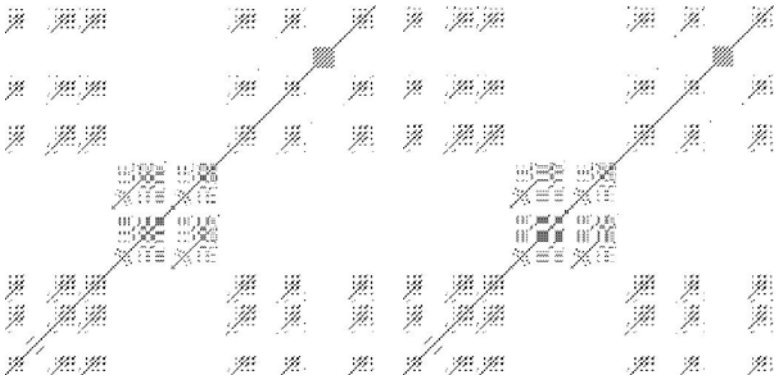

Fig. 7. Plots for the new and for the old component
Rys. 7. Wykres dla nowej i starej wersji komponentu

6. Summary

In this paper a new approach to analysis of logs of historic debuggers was proposed, described and examined. The proposed method aggregates information from a log being analysed and presents them in the graphical way. This kind of method of analysis is required because logs of historic debuggers could contain GBs of data [11] and it makes its

investigation difficult. It is also worth mentioning that RP can be generated easily and quickly in order to facilitate analysis.

It was shown that RP will detect repeating/recurring states in a signal generated based on logs of historic debuggers. It was also shown that RP are very vulnerable to changes in an input signal. These changes could be related to a bug in a program or to users' activity. Besides it was observed that RP are additive for a signal consisting of methods identifiers. If A is a signal of form A=a1a2...an, where *ai* is a set of items in this signal, then a plot generated for a signal B of form *B=a1a2...an+1* will contain a plot for the signal A. RP are also to some extent insensitive to order of actions performed by a user in an application. If A is a signal of form A=a1a2..., where *ai* is a set of items in this signal, then a plot generated for a signal B of form B=permutation(a1a2...) will contain fragments of a plot generated for the signal A.

Nonetheless, much more work is needed regarding this topic. The analysis of RP should be automated. The manual analysis of plots turned out not to be easy. It is especially difficult to map structures in a plot to what exactly happened in the program. Author thinks that RQA might help in this case. It is also important to find out the way to generate RP, based on historic debuggers logs, in such a way they would not be so vulnerable to changes in the input signal. The solution may be the application of other types of signals, a different algorithm of calculating distances or another approach to generation of plots. Finally, it is also necessary to elaborate more work regarding generation of RP for very long time series. Experiments showed that sampling is not a perfect solution. It seems that RP can also be used in the analysis of other logs, e.g. system event and performance logs [8].

BIBLIOGRAPHY

1. Wang L., Liu X., Song A., Xu L., Liu T.: An Effective Reversible Debugger of Cross Platform Based on Virtualization. International Conference on Embedded Software and Systems, 2009, p. 448÷453.
2. Koju T., Takada S., Doi N.: An efficient and generic reversible debugger using the virtual machine based approach. 1st ACM/USENIX International Conference on Virtual execution environments, 2005, p. 79÷88.
3. Jeffrey H. Joel.: Chaos game representation of gene structure. Nucleic Acids Research, 1990, p. 2163÷2170.
4. Bautista-Thompson E., Brito-Guevara R.: Classification of Data Sequences by Similarity Analysis of Recurrence Plot Patterns. Seventh Mexican International Conference on Artificial Intelligence, 2008, p. 111÷116.
5. Commandline Recurrence Plots, http://tocsy.pikpotsdam. de/commandline-rp.php.

6. Rybak T., Mosdorf R.: Computer Users Activity Analysis Using Recurrence Plot. International Conference on Biometrics and Kansei Engineering, 2009, p. 189÷194.
7. CRP Toolbox, http://tocsy.pik-potsdam.de/CRPtoolbox/.
8. Kubacki M., Sosnowski J.: Creating a knowledge database on system dependability and resilience, Control and Cybernetics, Vol. 42, No. 1, 2013, p. 287÷307.
9. Kennel M., Brown R., Abarbanel H.: Determining embedding dimension for phase-space reconstruction using a geometrical reconstruction. Physical Review A, Vol. 45, No. 6, 1992, p. 3403÷3411.
10. Lee J.: Dynamic Reverse Code Generation for Backward Execution. In: Proceedings of the Workshop on Verification and Debugging, Vol. 174, 2006, p. 37÷54.
11. Komorowski M.: Enhancing and extending IntelliTrace debugging capabilities. Studia Informatica, Vol. 33, No. 1, 2012.
12. Balzer R. M.: EXDAMS: extendable debugging and monitoring system. Proceedings of the May 14-16, 1969, Spring Joint Computer Conference, 1969, p. 567÷580.
13. Fraser A. M., Swinney H. L.: Independent coordinates for strange attractors from mutual information. Physical Review A, Vol. 33, No. 2, 1986, p. 1134÷1140.
14. IntelliTrace, http://msdn.microsoft.com/en-us/library/dd264915.aspx.
15. Recurrence Plots And Cross Recurrence Plots, http: //www.recurrence-plot.tk/.
16. Marwan N., Romano C., Thie M.l, Kurths J.: Recurrence plots for the analysis of complex systems. In: Physics Reports, Vol. 438, Issues 5–6, January 2007, p. 237÷329.
17. Eckmann J.P., Kamphorst S.O., Ruelle D.: Recurrence plots of dynamical systems, Europhys. Lett., Vol. 4, No. 9, 1987, p. 973÷977.
18. Chen S., Fuchs W. K., Chung J.: Reversible Debugging Using Program Instrumentation. IEEE Transactions on Software Engineering, Vol. 27, 2001, p. 715÷727.
19. Zalewski M.: Strange Attractors and TCP/IP Sequence Number Analysis, 2001, http://lcamtuf.coredump.cx/oldtcp/.
20. Saeed-Baginska A., Mosdorf R.: The recurrence plot as a tool in the analysis of network traffic anomaly detection. International Conference on Computer Information Systems and Industrial Management Applications, 2010, p. 289÷294.
21. Visual Recurrence Analysis, http://www.visualization-2002.org/VRA_ MAIN_PAGE_.html.

Omówienie

Wykres rekurencyjny to metoda analizy danych nieliniowych, polegająca na wytworzeniu wykresu (obrazka) pokazującego, kiedy stany analizowanego systemu, w różnych punktach czasowych, zbliżyły się do siebie.

Podejście to zostało z powodzeniem zastosowane do analizy danych finansowych, w geologii, w neurobiologii, a także w informatyce do analizowania ruchu sieciowego lub do analizy zachowań użytkowników.

W tym artykule opisano serię eksperymentów, w których zbadano możliwość zastosowania wykresów rekurencyjnych do analizowania danych zebranych przez debugery historyczne, np. do wykrywania błędów w aplikacji. W celu przeprowadzenia tych eksperymentów zaprojektowano i zaimplementowano narzędzie o nazwie *ExecutionTraceToolkit*, które wspomaga taką analizę.

Address

Michał KOMOROWSKI: Politechnika Warszawska, Instytut Informatyki, ul. Nowowiejska 15/19, 00-665 Warszawa, Polska, M.Komorowski@ii.pw.edu.pl/michalkomorowski@tlen.pl.