BugMaps-Granger: a tool for visualizing and predicting bugs using Granger causality tests

## Abstract

### Background

Despite the increasing number of **bug analysis tool**s for exploring bugs in software systems, there are no tools supporting the investigation of causality relationships between internal quality metrics and bugs. In this paper, we propose an extension of the **BugMaps** tool called **BugMaps-Granger** that allows the analysis of **source code** properties that are more likely to cause bugs. For this purpose, we relied on the **Granger Causality Test** to evaluate whether past changes to a given **time series** of source code metrics can be used to forecast changes in a **time series** of defects. Our tool extracts source code versions from **version control platforms**, calculates source code metrics and defects **time series**, computes **Granger Test results**, and provides interactive visualizations for causal analysis of bugs.

### Results

We provide an example of use of **BugMaps-Granger** involving data from the **Equinox Framework** and Eclipse **JDT Core** systems collected during three years. For these systems, the tool was able to identify the modules with more bugs, the average lifetime and complexity of the bugs, and the source code properties that are more likely to cause bugs.

### Conclusions

With the results provided by the tool in hand, a maintainer can perform at least two main **software quality assurance** activities: (a) refactoring the source code properties that Granger-caused bugs and (b) improving **unit tests coverage** in classes with more bugs.

## 1 Background

A number of software analysis tools has been proposed to improve software quality (Nierstrasz et al. [2005](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR18); Hovemeyer and Pugh [2004](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR14); Wettel [2009](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR24)). Such tools use different types of information about the structure and history of software systems. Basically, they are used to analyze software evolution, manage the quality of the source code, compute metrics, check coding rules, etc. In general, such tools help maintainers to understand large amounts of data coming from **software repositories**.

Particularly, there is a growing interest in analysis tools for exploring bugs in software systems (Hora et al. [2012](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR13); D’Ambros and Lanza [2012](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR8); Sliwerski et al. [2005](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR20); Dal Sassc and Lanza [2013](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR6)). Such tools help maintainers to understand the distribution, the evolutionary behavior, the lifetime, and the stability of bugs. For example, **Churrasco** is a web-based tool for collaborative software evolution analysis (D’Ambros and Lanza [2012](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR8)). The tool automatically extracts information from a variety of **software repositories**, including versioning systems and bug management systems. The goal is to provide an extensible tool that can be used to reason about software evolution under different perspectives, including the behavior of bugs. Other visualizations were also proposed for understanding the behavior of bugs, including system radiography (which provides a high-level visualization on the parts of the system more impacted by bugs) and bug watch (which relies on a watch metaphor to provide information about a particular bug) (D’Ambros et al. [2007](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR7)). **Hatari** (Sliwerski et al. [2005](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR20)) is a tool that provides views to browse through the most risky locations and to analyze the risk history of a particular component from a system. More recently, the tool in\*Bug (**Dal Sassc** and **Lanza** [2013](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR6)) was proposed to allow users navigating and inspecting the information stored in bug tracking platforms, with the specific purpose to support the comprehension of bug reports.

Despite the increasing number of **bug analysis tools**, they typically do not provide mechanisms for assessing the existence of correlations between the internal quality of a software system and the occurrence of bugs. To the best of our knowledge, there are no **bug analysis tools** that highlight the possible causes of bugs in the **source code**. More specifically, there are no tools designed to infer eventual causal relations between changes in the values of source code metrics and the occurrence of defects in **object-oriented classes**.

In this paper, we propose and describe the **BugMaps-Granger tool**—an extension of the **BugMaps tool** (Hora et al. [2012](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR13))—that supports detection of causal relations between source code metrics and bugs. The tool provides mechanisms to retrieve data from **software repositories**, to compute source code metrics, to generate time series of source code metrics and defects, and to infer causal relations between source code properties and defects. Moreover, **BugMaps-Granger** provides visualizations for identifying the modules with more bugs, the average lifetime and complexity of bugs, and the source code properties that are more likely to cause bugs. More specifically, our tool relies on the **Granger Causality Test** (**Granger** [1981](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR12)) to identify causal relations between time series of source code metrics and defects. This test evaluates whether past changes to a given time series of source code metrics can be used to forecast changes in a time series of defects. The proposed tool has the following features:

* The tool automatically extracts source code models of a target system from its **version control** platform in predefined time intervals.
* The tool generates time series of twelve source code metrics and time series with the number of defects in each class of the target system.
* The tool computes the **Granger Test** considering the metrics and defects time series to highlight possible causal relations.
* The tool integrates models extracted from the source code with models representing the number of bugs.
* The tool provides a set of interactive visualizations to support **software maintainers** in answering questions such as: (a) Which are the modules with more bugs? (b) What is the average lifetime of bugs? (c) What is the complexity of bugs? (d) What are the source code properties that Granger-cause bugs in a given module?, and (e) What are the metrics with the highest number of positive **Granger tests**?

The ultimate goal of **BugMaps-Granger** is to predict the changes in the source code that are more likely to cause defects. For example, with our tool in hand, a maintainer (before making a commit with changes to a given class) can verify whether such changes affect the values of source code metrics that, in the past, **Granger-caused defects**. If the changes significantly affect these metrics values, the maintainer can, for example, perform extra **software quality assurance** activities (e.g., she can conduct more unit testing or perform a detailed code inspection) before executing the **commit**.

In a previous conference paper, we described an exploratory study on using **Granger** to predict bugs (Couto et al. [2012](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR2)). Recently, this paper was extended with a concrete approach that relies on Granger Tests to trigger alarms whenever risky changes are applied in the **source code** (Couto et al. [2014](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR5)). A preliminary version of **BugMaps**—without any support to Granger Tests—is described in a short paper (Hora et al. [2012](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR13)).

## 2 Implementation

The execution of the **BugMaps-Granger tool** is divided into two phases: preprocessing and visualization. The preprocessing phase is responsible for extracting source code models, creating time series, and applying the Granger Test to compute possible causal relations between source code metrics and bugs. In the visualization phase, the user interacts with the tool. For example, he can retrieve the most defective classes of the system and visualize the source code properties that Granger-caused bugs in such classes.

**BugMaps-Granger** is implemented in Moose (Moose platform [2014](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR16)), which is a platform for software and data analysis (Nierstrasz et al. [2005](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR18)). Figure [1](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#Fig1) shows **BugMaps-Granger’s** architecture, which includes four modules: model extraction, time series creation, Granger Test module, and visualization module.

**Figure 1**

### 2.1 Model extraction

This module receives as input the **URL** associated to the **version control platform** of the target system (**SVN** or **Git**) and a time interval to be used in the analysis of the bugs. To extract the source code models, the module performs the following tasks: (a) it extracts the source code versions from the **version control platforms** in intervals of **bi-weeks**; (b) it removes test classes, assuming that such classes are implemented in directories and subdirectories whose name starts with the words “Test” or “test”; and (c) it parses the source code versions and generates **MSE** files using the **VerveineJ** tool (**Ducasse** et al. [2011](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR10); **VerveineJ** parser [2014](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR23)). **MSE** is the default file format supported by the **Moose** platform to persist source code models.

### 2.2 Time series creation

To create the time series of source code metrics, this module receives as input the models extracted by the previous module. For each class of each extracted model, the module relies on the **Moose platform** to compute eleven source code metrics including six **CK** metrics (proposed by **Chidamber** and **Kemerer** [1994](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR1)) and five others, such as lines of code, **FAN**-**IN**, **FAN-OUT**, etc. Table [1](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#Tab1) shows the source code metrics considered by the tool.

**Table 1 Source code metrics considered by BugMaps-Granger**

To create the time series of defects for each class, the module receives as input a **CSV** file containing the bugs (**IDs** and creation dates) collected from the bug tracking platforms (e.g., **Bugzilla, Jira, Mantis**, etc.). Basically, the module maps the bugs to their respective commits, using the mapping strategy presented in details in (Couto et al. [2012](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR2); Couto et al. [2014](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR5)). Next, the **source code** files changed by such commits are used to identify the classes changed to fix the respective bugs.

### 2.3 Granger test module

This module applies the **Granger Causality Test** considering the metrics and defects time series. To apply the **Granger Test**, the module relies on Algorithm 1. In this algorithm, Classes is the set of all classes of the system (line 1) and Defects[c] is the time series with the number of defects (line 2). The algorithm relies on function d\_check (line 3) to check whether the defects in the time series d conform to the following preconditions:

#### Algorithm 1 Applying the Granger Test

* P1: The time series must have at least 30 values. The motivation for this precondition is the fact that classes that only existed for a small proportion of the time frame considered in the analysis do not present a considerable history of defects to qualify their use in predictions.
* P2: The values in the time series of defects must not be all equal to zero. The motivation for this precondition is that it is straightforward to predict defects for classes that never presented a defect in their lifetime; probably, they will remain with zero defects in the future.
* P3: The time series of defects must be stationary, which is a precondition required by the **Granger Test** (Fuller [1995](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR11)).

Suppose that a given class c passed the previous preconditions. For this class, suppose also that M[n][c] (line 5) is the time series with the values of the n-th considered source code metric, 1≤n ≤NumberOfMetrics. The algorithm relies on function m\_check (line 6) to test whether time series m—a time series with metrics values—conforms to the following preconditions:

* P4: The time series of source code metrics must not be constant. In other words, metrics time series whose values never change must be discarded, since variations in the independent variables are the key event to observe when computing Granger causality.
* P5: The time series of source code metrics must be stationary, as defined for the **defects series.**

Finally, for the time series m (source code metrics) and d (defects) that passed preconditions P1 to P5, function granger(m,d) checks whether m **Granger-causes** d (line 7). In practice, to apply the test, BugMaps-Granger relies on the function *granger.test()* provided by the *msbvar* (**MSBVAR** package [2012](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR17)) package of the R system.

It is worth mentioning that we previously performed an extensive study to evaluate the application of **Granger Causality Test** on software defects prediction (Couto et al. [2014](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR5)). Basically, we focus on answering questions such as: (a) How many time series pass the preconditions related to defects (preconditions P1, P2, P3)? (b) How many time series pass the preconditions related to source code metrics (preconditions P4 and P5)? (c) How many classes present positive results on the **Granger Test**? (d) What is the number of defects potentially covered by Granger? To answer these questions, we used a dataset including time series of source code metrics and defects for four real-world systems (**Eclipse JDT Core, Eclipse PDE UI, Equinox Framework**, and **Lucene**) (Couto et al. [2013b](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR4)).

### 2.4 Visualization module

This module receives the following input data: a file containing the bugs mapped to their respective classes and the **Granger** results, a model extracted from the last source code version, and the source code itself of the system under analysis. From this information, the module provides four interactive visualization **browsers**:

* Two **browsers** are used for analysis. The first one deals with the classes, the number of bugs, and the Granger results of the system under analysis (called **Granger browser**) while the second one deals with the complexity of the bugs (called Bug **as Entity browser**).
* Two browsers are used to rank the classes and the metrics most involved with bugs.

Such browsers are implemented using visualization packages provided by the **Moose** Platform. Basically, the visualizations are based on **Distribution Map**, a generic technique to reason about the results of software analysis and to investigate how a given phenomenon is distributed across a software system (**Ducasse** et al. [2006](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#ref-CR9)). Using a **Distribution Map**, three metrics can be displayed through the height, width, and color of the objects in the map. In our maps, rectangles represent classes or bugs and containers represent packages.

Figure [2](https://jserd.springeropen.com/articles/10.1186/2195-1721-2-1#Fig2) shows the **Granger** browser, which has four panes: visualization of classes and packages (top left), measures (top right), Granger results (bottom left), and source code (bottom right)a. Metrics, source code, and Granger results are updated according to the selected class in the classes and packages pane.

1. Granger Causality Test (GCT) - тест на причинно-следственные связи Грэнджера
2. BugMaps-Granger - инструмент для визуализации и прогнозирования ошибок с использованием тестов на причинно-следственные связи Грэнджера
3. Software Quality Assurance (SQA) - обеспечение качества программного обеспечения
4. Equinox Framework - рабочий каркас Equinox
5. Eclipse JDT Core - основная часть Eclipse Java Development Tools
6. Source Code Metrics - метрики исходного кода
7. Version Control Platform - платформа управления версиями
8. Time Series - временные ряды
9. Defects - дефекты
10. Bug Analysis Tools - инструменты анализа ошибок
11. Object-Oriented Classes - объектно-ориентированные классы
12. Unit Tests Coverage - покрытие модульными тестами
13. Software Evolution - эволюция программного обеспечения
14. Bug Tracking Platforms - платформы отслеживания ошибок
15. Moose Platform - платформа Moose
16. SVN (Subversion) - система контроля версий
17. Git - распределенная система контроля версий
18. CSV (Comma-Separated Values) - значения, разделенные запятыми
19. MSE (Moose Source Extractor) - извлекатель исходного кода Moose
20. FAN-IN - количество методов, вызывающих данный метод
21. FAN-OUT - количество методов, вызываемых из данного метода
22. CK Metrics - метрики Чидамбера и Кемерера
23. Algorithm - алгоритм
24. Bi-weeks - двухнедельные интервалы
25. VerveineJ - инструмент VerveineJ
26. Lucene - поисковый движок
27. Time Frame - временной интервал
28. R System - система R
29. msbvar - пакет msbvar
30. Preprocessing - предварительная обработка
31. Visualization - визуализация
32. Model Extraction - извлечение модели
33. Test Classes - тестовые классы
34. CK Metrics - метрики CK
35. Visualization Browsers - визуализационные браузеры
36. Distribution Map - карта распределения
37. Bug as Entity Browser - браузер для анализа ошибок как сущностей
38. Source Code Parsing - разбор исходного кода
39. Granger Test Results - результаты теста Грэнджера
40. Moose Visualization Packages - пакеты визуализации Moose
41. Complexity of Bugs - сложность ошибок
42. Granger Browser - браузер Грэнджера
43. Moose Platform - платформа Moose
44. Moose Source Code Models - модели исходного кода Moose
45. Lucene - поисковый движок
46. Churrasco - инструмент Churrasco
47. Hatari - инструмент Hatari
48. Object-Oriented Systems - объектно-ориентированные системы
49. Software Repository - репозиторий программного обеспечения
50. Source Code Refactoring - рефакторинг исходного кода