Couto et al. Journal of Software Engineering Research and Development 2014, 2:1
www.jserd.com/content/2/1/1

Journal of Software Engineering Research and Development
a SpringerOpen Journal

SOFTWARE Open Access

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

Cesar Couto[1,2*], Marco Tulio Valente[2], Pedro Pires[2], Andre Hora[3], Nicolas Anquetil[3] and Roberto S Bigonha[2]

*Correspondence:
cesar@decom.cefetmg.br
[1]Department of Computer Science, UFMG, Belo Horizonte, Brazil
[2]Department of Computing, CEFET-MG, Belo Horizonte, Brazil
Full list of author information is available at the end of the article

Abstract

Background: Despite the increasing number of bug analysis tools 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.

Keywords: Bug analysis tools; Software metrics; Causality tests

1 Background

A number of software analysis tools has been proposed to improve software quality (Nierstrasz et al. 2005; Hovemeyer and Pugh 2004; Wettel 2009). 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; D'Ambros and Lanza 2012; Sliwerski et al. 2005; Dal Sassc and Lanza 2013). 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).

Springer

© 2014 Couto et al.; licensee Springer. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly credited.

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). Hatari (Sliwerski et al. 2005) 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) 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)—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) 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). 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). A preliminary version of BugMaps—without any support to Granger Tests—is described in a short paper (Hora et al. 2012). Later, we proposed a second version of this tool, which we called BugMaps-Granger, including support to Granger Causality (Couto et al. 2013a). In the present paper, we extend this initial work on BugMaps-Granger by including a more detailed presentation on the tool and a case study, with two large open-source systems (Eclipse JDT Core and Equinox Framework).

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), which is a platform for software and data analysis (Nierstrasz et al. 2005). Figure 1 shows BugMaps-Granger's architecture, which includes four modules: model extraction, time series creation, Granger Test module, and visualization module.

In the following subsections, we describe the modules of this architecture:

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

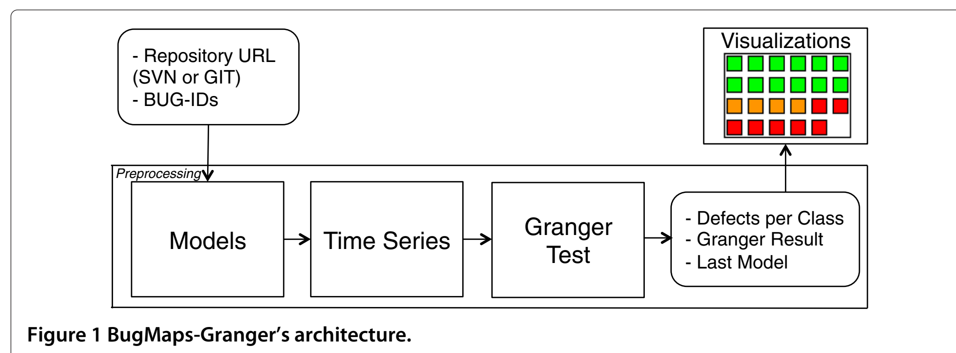

Figure 1 BugMaps-Granger's architecture.

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; VerveineJ parser 2014). 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) and five others, such as lines of code, FAN-IN, FAN-OUT, etc. Table 1 shows the source code metrics considered by the tool. In a second step, this module stores the computed metric values in CSV files. Basically, for a target system S and a metric M, this module creates a CSV file whose lines represent the classes of S and the columns represent the bi-weeks considered when extracting the versions of S. A cell (c, t) in this file contains the value of the metric M, measured for the class c, in the bi-week t.

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; Couto et al. 2014). 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

```
1:  for all c ∈ Classes do
2:      d = Defects[c];
3:      if d_check(d) then
4:          for n = 1 to NumberOfMetrics do
5:              m = M[n][c];
6:              if m_check(m) then
7:                  granger(m, d);
8:              end if
9:          end for
10:     end if
11: end for
```

Table 1 Source code metrics considered by BugMaps-Granger

	Metrics	Description
1	WMC	Weighted methods per class
2	DIT	Depth of inheritance tree
3	RFC	Request for class
4	NOC	Number of children
5	CBO	Coupling between object class
6	LCOM	Lack of cohesion in methods
7	FAN-IN	Number of classes that reference a given class
8	FAN-OUT	Number of classes referenced by a given class
9	NOA	Number of attributes
10	LOC	Number of lines oxf code
11	NOM	Number of methods

- 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).

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, $1 \leq n \leq$ `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) 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). 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).

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). 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 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.

Figure 3 shows the Bug as Entity browser which is composed by two panes: visualization of classes and packages (left pane) and bugs (right pane). When a defective class is selected, the bugs in the class are colored in black (in the right pane). In contrast, when a bug is selected, the classes changed to fix this bug are colored in black (in the left pane). BugMaps-Granger also shows a list of classes ranked by the number of defects and the number of Granger tests with a positive result.

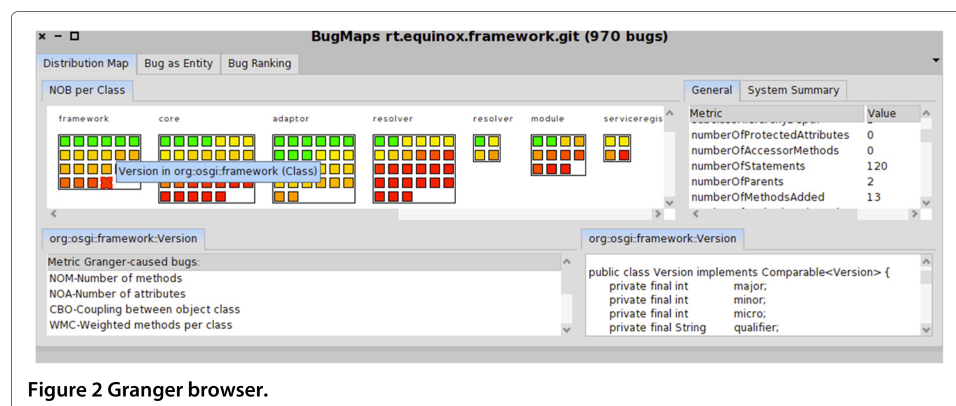

Figure 2 Granger browser.

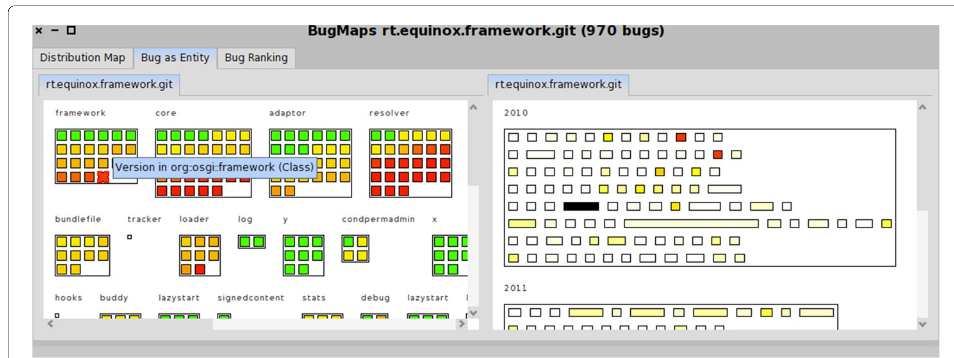

Figure 3 Bug as Entity browser.

3 Results and discussion

In this section, we provide an example of use considering data from the Equinox Framework and Eclipse JDT Core systems collected during three years. For Equinox Framework, the tool extracted 79 source code versions in intervals of bi-weeks, including 417 classes, from 2010-01-01 to 2012-12-28. For Eclipse JDT Core, the tool extracted 78 source code versions in intervals of bi-weeks, including 2,467 classes, from 2005-01-01 to 2007-12-15. In a second step, for each class, the tool created eleven time series of source code metrics (for each metric in Table 1) and one time series of defects. Finally, for each pair of time series (source code metrics and defects), the tool applied the Granger Test to identify causal relations. We analyzed the Granger results for both systems according to the proposed visualizations, as discussed next.

3.1 Granger

In this map, the rectangles are the classes of the target system, as illustrated in Figure 4. The color of a class represents the number of bugs detected through its history ranging from green to red (the closer to red, more bugs the class had in its history). By selecting a defective class, the bottom pane is updated showing the source code metrics that Granger-caused bugs in this class. Particularly, Figure 4 provides an overview of the distribution of the bugs in the Equinox Framework system. We can observe that the `resolver` package contains a significant number of classes with bugs. Moreover, for the class `org.eclipse.osgi.internal.resolver.StateImpl`, we can observe

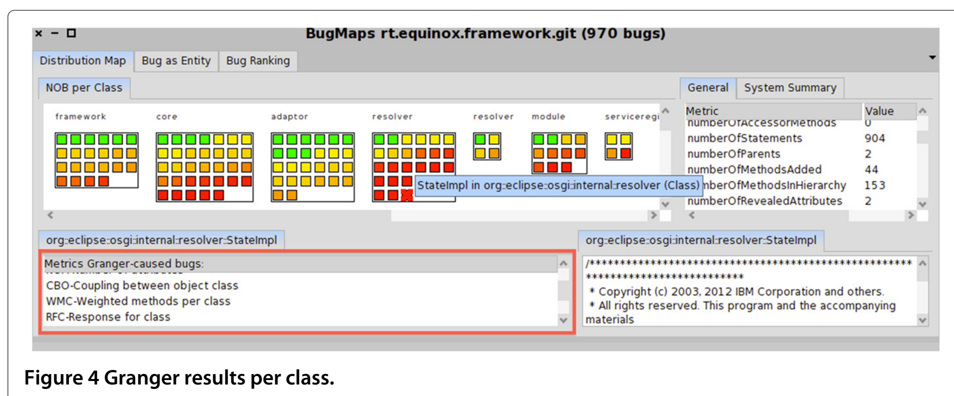

Figure 4 Granger results per class.

that the source code metrics that Granger-caused bugs were CBO (Coupling Between Object), WMC (Weighted Methods per Class), and RFC (Response for Class).

The Granger browser can also be used to avoid future defects. For example, with this result in hand, a maintainer (before making a commit with changes to the `StateImpl` class) can verify whether such changes heavily affect the values of source code metrics that Granger-caused defects in the past (in our example, the metrics that Granger-caused defects were CBO, WMC, and RFC). If the change affects these metrics, the maintainer can for example perform extra software quality assurance activities in this class (like unit testing or code inspection).

3.2 Bug as entity

As illustrated in Figure 5, this map represents bugs instead of classes. The color of a bug represents its lifetime, i.e., the number of days the bug remained opened. Blue denotes a bug that was still open at the end of the considered time period. Moreover, white denotes a bug that was open for a short time. Similarly, yellow is used for a bug that was open up to three months, and red for a bug that was opened for more than three months. The width of a bug representation denotes its complexity, measured as the number of classes changed to fix the bug. Bugs are sorted according to the date they were created.

Figure 5(a) shows the bugs of the Equinox Framework created in 2010. We can observe that all bugs from 2010 were fixed (i.e., there are no bugs in blue), that only two bugs remained open for more than three months (bugs going to red), and that complex bugs (long width) are dispersed in time. Figure 5(b) shows the bugs of the Eclipse JDT Core created in 2005. Similar to the Equinox Framework, all bugs were fixed and few bugs remained open for more than three months. In addition, most bugs have low complexity (short width). However, in a detailed analysis, we can also observe that the highlighted bug (ID 89096) is quite complex. More specifically, the developer team changed 75 classes in

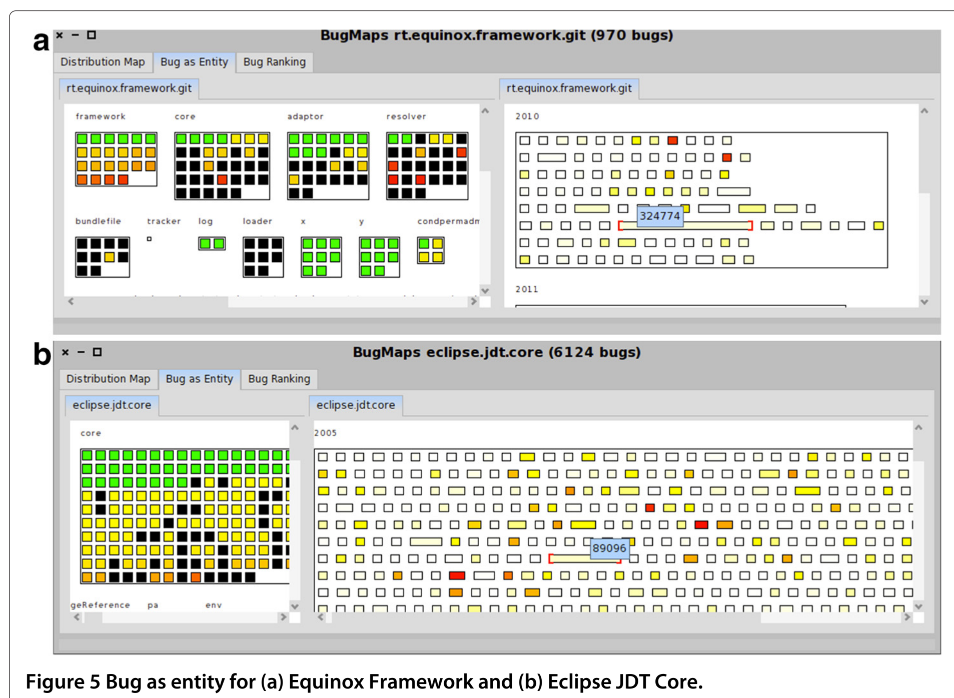

Figure 5 Bug as entity for (a) Equinox Framework and (b) Eclipse JDT Core.

order to fix this particular bug, which is related to a performance problem in the resource bundle mechanism (a requirement scattered by the classes of the JDT Core).

3.3 Bug ranking

Figure 6 shows examples of the Bug Ranking browser which is composed by one grid with two columns: class names (first column) and number of defects during the time frame considered in the analysis (second column). The figure shows the classes ranked by the defects for the Equinox Framework and Eclipse JDT Core systems during the period of three years. For the Equinox Framework, the two classes with more defects are `osgi.internal.module.ResolverImpl` and `osgi.framework.internal.core.Framework`. The `ResolverImpl` class—which has more than 2,000 lines of code—is an important class responsible for resolving the constraints of the bundles (JAR components) in a system that follows the OSGi standard (Tavares and Valente 2008). The `Framework` class also has more than 2,000 lines of code and represents the core OSGi Framework class. For Eclipse JDT Core, the top-ten classes with more defects include classes such as `org.eclipse.jdt.core.dom.AST` and `org.eclipse.jdt.internal.core.JavaModel`. The `AST` class represents an abstract syntax tree node factory and the `JavaModel` class is responsible for managing projects in the Eclipse workspace.

3.4 Granger ranking

For each valid time series of source code metrics, Figure 7 shows the number of Granger tests that returned a positive result. For example, the number of CBO time series with a

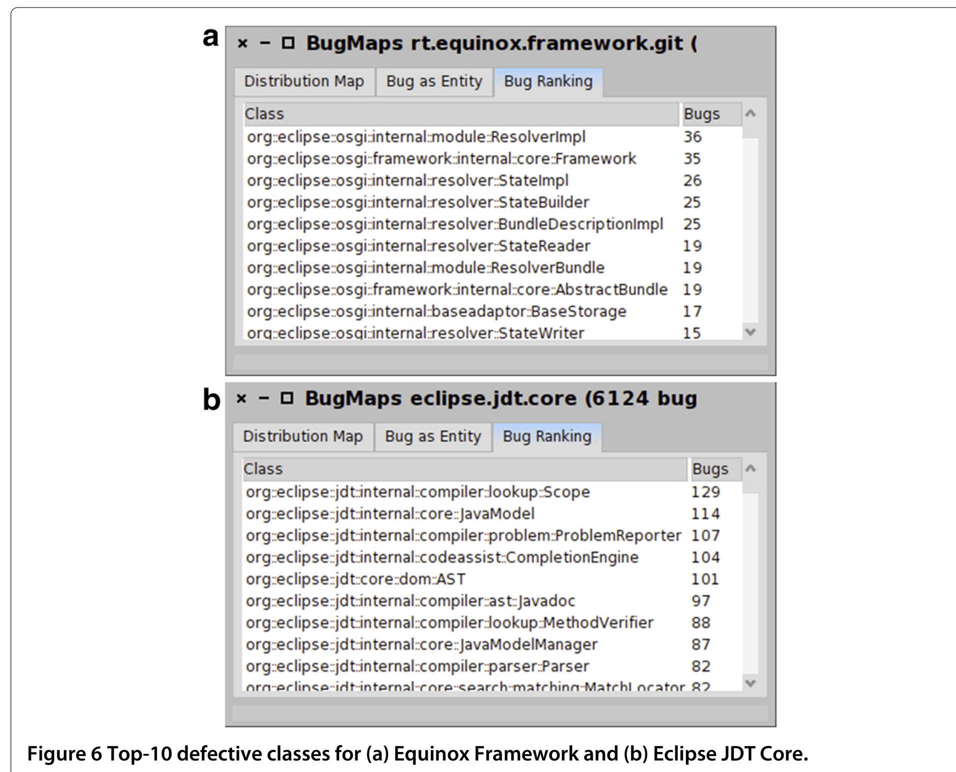

Figure 6 Top-10 defective classes for (a) Equinox Framework and (b) Eclipse JDT Core.

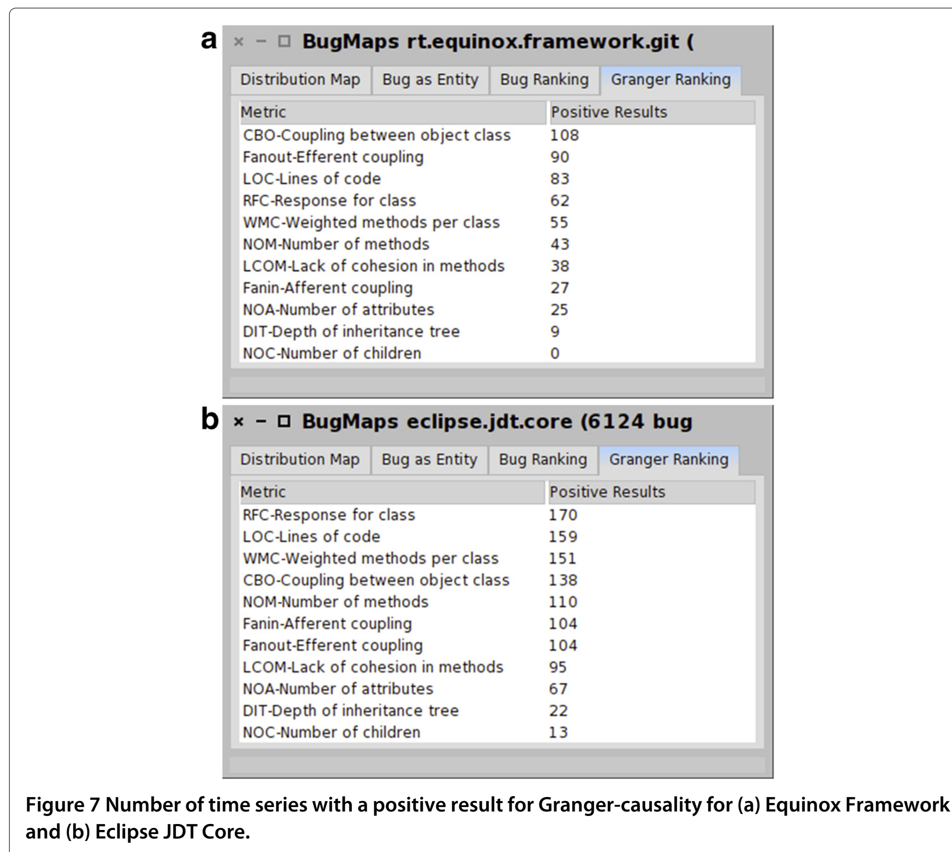

Figure 7 Number of time series with a positive result for Granger-causality for (a) Equinox Framework and (b) Eclipse JDT Core.

Granger-causality with defects was 108 and 138 series for Equinox Framework and Eclipse JDT Core, respectively. As can be observed, the metrics with the highest number of positive Granger tests were CBO, FAN-OUT, LOC, RFC, and WMC for Equinox Framework; and RFC, LOC, WMC, and CBO for Eclipse JDT Core. On the other hand, the metrics with the lowest number of positive results were DIT and NOC for both systems.

Based on these results, we can conclude that metrics related to complexity (WMC), coupling (CBO, RFC, and FAN-OUT), and LOC tend to impact in the occurrence of defects in the Equinox Framework and Eclipse JDT Core systems, at least according to Granger. Conversely, metrics related to inheritance—such as DIT and NOC—tend to have a small influence in the occurrence of defects.

4 Conclusion

In this paper, we described a tool that infers and provides visualizations about causality relations between source code metrics and bugs. The BugMaps-Granger tool extracts time series of defects from such systems and allows the visualization of different bug measures, including the source code properties that Granger-caused bugs. The ultimate goal of BugMaps-Granger is to highlight changes in the source code that are more subjected to bugs, and the source code metrics that can be used to anticipate the occurrence of bugs in the changed classes. With this tool in hand, maintainers can perform at least two main actions for improving software quality: (a) refactoring the source code properties that Granger-caused bugs and (b) improving unit tests coverage in classes with more bugs.

As future work, we intend to extend BugMaps-Granger with other internal software quality metrics, including metrics associated to violations in the static architecture of software systems, as revealed by the DCL language (Terra and Valente 2009) or the ArchLint tool (Maffort et al. 2013), for example. Another possible research thread concerns the relations between defects and code smells. In this case, we intend to start by investigating the relations between defects and methods located in inappropriate classes (i.e., feature envy instances), as revealed by the JMove recommendation system (Sales et al. 2013). In addition, we plan to extend BugMaps-Granger with a new functionality for alerting maintainers about the future occurrence of defects. We intend to implement this tool as a plug-in for version control platforms, like SVN and Git. Basically, this new tool should trigger alarms whenever risky changes are committed to version control platforms.

5 Availability and requirements

To execute BugMaps-Granger, the requirements of the target system are:

- Java-based systems.
- Identifiers and creation dates of bugs stored in a CSV file.
- URL or directory path of the version control platforms (SVN or GIT).

Additional information about BugMaps-Granger:

- **Project name:** BugMaps-Granger.
- **Project home page:** http://aserg.labsoft.dcc.ufmg.br/bugmaps/.
- **Operating system(s):** MacOS, Linux, and Windows.
- **Programming language:** Java, Smalltalk, and R.
- **License:** BugMaps-Granger is an open source project, distributed under a MIT license.

Endnote

[a] Since most of our visualizations make heavy use of colors, we provide high-resolution versions of these figures in a companion website: http://aserg.labsoft.dcc.ufmg.br/bugmaps.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

CC and MTV worked on the design and implementation of the tool and also wrote this manuscript. PP worked on the implementation of the preprocessing and visualization modules of the tool. AH and NA participated in the implementation of the first version of the tool (Hora et al. 2012). In this second version, they provided technical support on the Moose platform and revised the manuscript. Finally, RSB revised the manuscript. All authors read and approved the final manuscript.

Acknowledgements

This research is supported by grants from FAPEMIG, CNPq, and CAPES (Brazil) and INRIA (France).

Author details

[1]Department of Computer Science, UFMG, Belo Horizonte, Brazil. [2]Department of Computing, CEFET-MG, Belo Horizonte, Brazil. [3]RMoD Team, INRIA, Lille, France.

Received: 20 November 2013 Accepted: 27 February 2014
Published: 21 March 2014

References

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6): 476–493

Couto C, Silva C, Valente MT, Bigonha R, Anquetil N (2012) Uncovering causal relationships between software metrics and bugs. In: 16th European Conference on Software Maintenance and Reengineering (CSMR). IEEE Computer Society, USA, pp 223–232
Couto C, Pires P, Valente MT, Bigonha R, Anquetil N (2013a) BugMaps-Granger: A Tool for Causality Analysis between Source Code Metrics and Bugs. Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Session, Brazilian Computer Society, Brazil
Couto C, Maffort C, Garcia R, Valente MT (2013b) COMETS: A Dataset for Empirical Research on Software Evolution Using Source Code Metrics and Time Series Analysis. ACM SIGSOFT Softw Eng Notes 38(1): 1–3
Couto C, Pires P, Valente MT, Bigonha R, Anquetil N (2014) Predicting software defects with causality tests. J Syst Soft. doi: http://dx.doi.org/10.1016/j.jss.2014.01.033
Dal Sassc T, Lanza M (2013) A closer look at bugs. In: 1st Working Conference on Software Visualization (VISSOFT). IEEE Computer Society, USA, pp 1–4
D'Ambros M, Lanza M, Pinzger M (2007) A bug's life: Visualizing a bug database. In: 4th International Workshop on Visualizing Software for Analysis and Understanding (VISSOFT). IEEE Computer Society, Canada, pp 113–120
D'Ambros M, Lanza M (2012) Distributed and collaborative software evolution analysis with churrasco. Sci Comput Program 75(4): 276–287
Ducasse S, Girba T, Kuhn A (2006) Distribution Map. In: 22nd International Conference on Software Maintenance (ICSM). IEEE Computer Society, USA, pp 203–212
Ducasse S, Anquetil N, Bhatti MU, Hora A, Laval J, Girba T (2011) MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Technical report, RMOD - INRIA Lille - Nord Europe, Software Composition Group - SCG
Fuller WA (1995) Introduction to Statistical Time Series. John Wiley & Sons, USA, pp 546–663
Granger C (1981) Some properties of time series data and their use in econometric model specification. J Econometrics 16(6): 121–130
Hora A, Couto C, Anquetil N, Ducasse S, Bhatti M, Valente MT, Martins J (2012) Bugmaps: A tool for the visual exploration and analysis of bugs. In: 16th European Conference on Software Maintenance and Reengineering (CSMR Tool Demonstration). IEEE Computer Society, USA
Hovemeyer D, Pugh W (2004) Finding bugs is easy. SIGPLAN Notices 39(12): 92–106
Maffort C, Valente MT, Anquetil N, Hora A, Bigonha M (2013) Heuristics for discovering architectural violations. In: 20th Working Conference on Reverse Engineering (WCRE). IEEE Computer Society, USA, pp 222–23
Moose platform (2014). http://www.moosetechnology.org
MSBVAR package (2012). http://cran.r-project.org/web/packages/MSBVAR/index.html
Nierstrasz O, Ducasse S, Grba T (2005) The story of Moose: an agile reengineering environment. In: 10th European Software Engineering Conference (ESEC). ACM, USA, pp 1–10
Sales V, Terra R, Miranda LF, Valente MT (2013) Recommending move method refactorings using dependency sets. In: 20th Working Conference on Reverse Engineering (WCRE). IEEE Computer Society, USA, pp 232–241
Sliwerski J, Zimmermann T, Zeller A (2005) Hatari: Raising risk awareness. In: 10th European Software Engineering Conference (ESEC). ACM, USA, pp 107–110
Tavares A, Valente MT (2008) A gentle introduction to OSGi. ACM SIGSOFT Softw Eng Notes 33(5): 1–5
Terra R, Valente MT (2009) A dependency constraint language to manage object-oriented software architectures. Softw: Pract Exp 32(12): 1073–1094
VerveineJ parser (2014). http://www.moosetechnology.org/tools/verveinej
Wettel R (2009) Visual exploration of large-scale evolving software. In: 31st International Conference on Software Engineering (ICSE). IEEE Computer Society, USA, pp 391–394

doi:10.1186/2195-1721-2-1
Cite this article as: Couto *et al.*: **BugMaps-Granger: a tool for visualizing and predicting bugs using Granger causality tests.** *Journal of Software Engineering Research and Development* 2014 **2**:1.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Immediate publication on acceptance
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com