Automatic Event Detection for Software Product Quality Monitoring

Dennis Bijlsma
Software Improvement Group
Amsterdam, The Netherlands
Email: d.bijlsma@sig.eu

José Pedro Correia
Software Improvement Group
Amsterdam, The Netherlands
Email: zepedro.correia@gmail.com

Joost Visser
Software Improvement Group &
Radboud University Nijmegen
The Netherlands
Email: j.visser@sig.eu

***Abstract*—Collecting product metrics during development or maintenance of a software system is an increasingly common practice that provides insight and control over the evolution of a product's quality. An important challenge remains in interpreting the vast amount of data as it is being collected and in transforming it into actionable information.**

We present an approach for discovering significant events in the development process from the associated stream of product measurement data. At the heart of our approach lies the view of measurement data streams as functions for which derivatives can be calculated. In a manner inspired by Statistical Process Control, a certain number of data points are then selected as events worthy of further inspection.

We apply our approach in an industrial setting, namely as support to the Software Monitoring service provided by the Software Improvement Group. In particular, we report on an evaluation of an alert service that continuously checks for events in over 400 monitored software systems.

***Index Terms*—Event Detection, Data Streams, Statistical Process Control, Software Product Quality**

I. Introduction

Software product measurement is an established field of software engineering. A large number of software metrics have been developed in the past decades (for an overview, see [7]) and research continues to be done in this area. Platforms that support continuous measurement are starting to emerge, such as Sonar[1] or Alitheia Core[2].

Measurement is not sufficient by itself though, requiring means of analysis and interpretation of the data, as well as a methodology to bind everything together. The Software Improvement Group (SIG) has developed such a methodology for monitoring the technical quality of software products [13], [5]. It is grounded on software measurement, uses a hierarchical aggregation model to summarize the data, and follows a standardized analysis and reporting process. The source code of the monitored software system is analyzed periodically (typically every week) and, at a lower frequency (typically every three months), a software engineering expert is responsible for interpreting the data and communicating his findings to the customer.

[1]http://sonar.codehaus.org/
[2]http://www.sqo-oss.org/xwiki/bin/view/Main/

The process has, however, two limitations. Firstly, the feedback time might be too long. A situation might arise that calls for prompt action from the customer, so the feedback given at a low frequency might not be timely enough. Secondly, the scalability of the approach (more systems and/or more metrics) is hindered by the human limitations of the experts in keeping track of events.

The work presented describes a method for automatically detecting significant events in a set of measurement streams associated with a monitored software product.

This paper is structured as follows. In Section II we describe the setting, i.e., measurement-based software product monitoring and how automatic event detection is valuable in that context. Sections III and IV formalize the problem of detecting events in a single measurement stream and describes our approach. In Section V and VI we describe the application of the event detection algorithm, and report on a quantitative evaluation of its performance when used to monitor a large number of software systems under active development or maintenance. This is further discussed in Section VII. Finally, in Section VIII we discuss related work and we finish with conclusions and future work in Section IX.

II. Background

A. The SIG Maintainability Model

The SIG performs assessment of software product quality (in terms of maintainability) based on source code analysis. Firstly, values for a set of well-known metrics are calculated using automated tools. These include lines of code (LOC), duplicated LOC, McCabe complexity, parameter counts, and dependency counts. Measurements are performed at different levels of granularity, such as lines, units (e.g. methods or functions), and modules (e.g. files or classes).

Secondly, the low-level results are aggregated into language independent ratings for system-level properties. These ratings convey qualitative assessments of the measurements. At the time of writing, the following properties are being assessed:

Volume — The larger a system, the more effort it takes to maintain since there is more information to be taken into account;

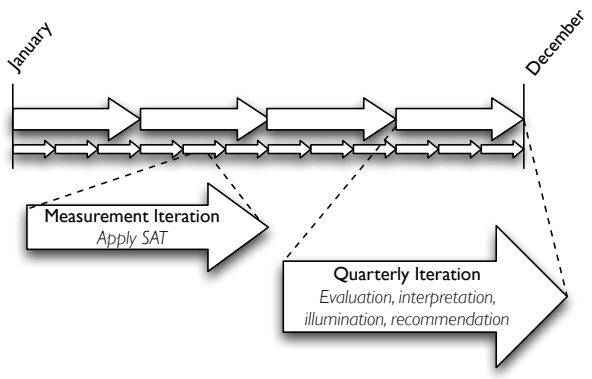


Fig. 1. Nested iterations in the process of software monitoring at SIG.

Redundancy	Duplicated code has to be maintained in all places where it occurs;
Unit size	Units, as the lowest-level piece of functionality, should be kept small in order to be specialized and easier to understand;
Complexity	Simple systems are easier to comprehend and test than complex ones;
Unit interface	Units with many parameters can be a symptom of bad encapsulation;
Coupling	Tightly coupled components are more resistant to change, since modifications in one component might entail modifications in components that depend on it.

Thirdly, the ratings for these properties are aggregated into ratings for higher-level concepts based on the ISO/IEC 9126 [12]. The SIG Maintainability Model was originally described by Heitlager *et al.* [10] but has subsequently gone through some modifications. Baggen *et al.* [3] provide a more detailed and up to date description of the model. Alves *et al.* [2], [1] elaborate on its scientific underpinnings.

B. Software product monitoring

One of the applications of the SIG Maintainability Model is in supporting a Software Monitoring service [13], [5]. This service provides high-level information regarding the quality of software systems to management-level people. It is organized mainly in two nested iterations (illustrated in Figure 1):

- Measurement iteration: On a periodical basis (typically each week), the producer/maintainer of the software system provides the SIG with a snapshot of the source code, which is then subjected to a battery of static analysis algorithms that compute measurements. These values are then stored and aggregated according to the SIG Maintainability Model and a measurement report becomes available. The snapshot consist only of source code, the SIG does not have access to the software system's source code repository or issue tracking system.
- Quarterly iteration: Every three months, the technical data gathered in the measurement iterations is analyzed and interpreted by one or more software engineering experts. Their task is two-fold, namely to: 1) detect and investigate outstanding events, trends or other signs in the data that indicate potential problems; 2) interpret the detected potential issues in the business context of the software system and provide actionable information and recommendations to the customer.

This process has two important limitations:

- Responsiveness: In the current setting, the time it takes for the customer to receive feedback regarding a quality issue in the monitored software system can go up to three months. This means that, when the issue gets reported, it may not be relevant anymore, be already fixed or be very difficult to address at that point. Ideally one would like to keep the customer informed of any quality issue introduced as soon as it becomes evident;
- Scalability: For the quarterly iteration, the software engineering expert needs to look back into the measurement streams for that period in order to detect relevant events. The number of data points is typically large, making the analysis more time-consuming and error prone. One would like to automate part of it in order to allow the experts to focus on the interpretation of the events, rather than on their detection.

To address these two limitations, an automatic alert service was created to support the experts.

III. The alert service

As described above, on arrival, each new snapshot is subject to measurement using static analysis algorithms. The results are then aggregated according to the SIG Maintainability Model into *ratings*.

For the alert service, we chose to apply the event detection method to the streams of ratings for the 6 basic properties of the model (see Section II). The reason not to track the measurement streams directly is that ratings provide some advantages, namely:

- Some ratings are calculated using *risk profiles*[3], which summarize a distribution (for example, the McCabe complexity per unit) in a more informative way than an average would, thus allowing for more sensitivity;
- Every rating is a value in the $[0.5, 5.5[$ interval, thus the same thresholds can be used for all of them.

On each new snapshot of a given system, in case events are detected for any of the 6 monitored properties, an alert email is automatically sent to the experts involved

[3] Risk profiles characterize a metric, for example unit size, in terms of the percentage of lines of code in units classified as low, medium, high or very high risk. This classification is based on thresholds for the given metric, defined based on the literature and calibrated against a large set of systems. See Alves *et al.* [2] for more details.

```
For the snapshot dated 2010-03-12 the follow-
ing quality highlights were detected:

Deteriorations:
- Duplication by -0.22 (abrupt)

Improvements:
- Unit size by +0.37 (abrupt)
- Unit complexity by +0.61 (sus-
tained since 2009-12-15)
```

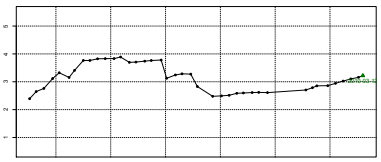


Fig. 2. Example of an alert email with chart for unit complexity (the two other charts are omitted).

in monitoring that system. This email contains some information to identify the system and the snapshots analyzed, as well as data regarding the events detected. These are grouped as *improvements* and *deteriorations*, i.e., events triggered by an increase in a rating or a decrease, respectively. Furthermore, they are distinguished as *abrupt*, which capture fast and sudden variations, or *sustained*, which consist of continuous change in a certain direction. For the former, the difference in value from the previous snapshot is shown. For sustained change events, the accumulated value of the increase/decrease of the rating is shown, together with the date when the trend was initiated. These are accompanied by charts of the ratings through time, tagged with the data points were events were detected. An example can be seen in Figure 2.

The main purpose of these emails is to notify the experts monitoring the system that there has been a noteworthy change in the system in comparison to the previous measurement iteration. This explains wny the email's contents focuses more on the changes in rating than on the absolute values. The emails are not supposed to identify the cause of the alert. If the recipient of the email considers the alert to be important enough he can use the underlying measurement to identify the reason of the rating change.

A. The general scenario

Although the alert service described here specifically fits the SIG Maintainability Model and the Software Monitoring service, one can take a step back and look at the general scenario. The methodology for event detection can be applied in any scenario where a software product is continuously monitored using source code measurement.

In the general scenario, snapshots of the source code are available on a periodical basis. Each of these snapshots is subject to a set of algorithms that each computes a value for a metric. The result is, for each metric m, a periodically updated set of measurements per snapshot, which we will refer to as a *measurement stream* ζ^m.

The objective is to process each measurement stream ζ^m with an event detection algorithm that would reduce it to a sequence of events Ω^m. These can then be aggregated into a report for a software engineering expert, available either periodically or on-demand, depending on application and context.

In the following section we describe the algorithm used to detect events in a measurement stream. Since the description regards a measurement stream for a single metric m, for simplicity of notation the superscript m will be omitted.

IV. Event detection

Let ζ be a measurement stream defined as $\{(x_0, t_0), \ldots, (x_n, t_n)\}$ where $x_i \in \mathbb{R}$ is the value of a measurement and t_i the physical time when it was obtained.

For $(x_i, t_i), (x_j, t_j) \in \zeta$, the indices $i, j \in \mathbb{N}_0$ represent logical time and t_i, t_j physical time, where $i < j \Leftrightarrow t_i < t_j$ (i.e. the logical order in ζ corresponds to a natural order with respect to physical time).

What we want to calculate is a set of events $\Omega \subseteq \mathbb{N}$ with indices of the data points that are considered of interest. We take into account two different situations that can trigger events, namely *abrupt change* and *sustained change*. Thus, $\Omega = \Omega_a \cup \Omega_s$, where Ω_a is the set of abrupt change events and Ω_s the set of sustained change events. The calculation of Ω_s makes use of the values from Ω_a, thus the latter needs to be calculated before the former.

A. Abrupt change

We define abrupt change as a rapid and unexpected increase or decrease in a measurement stream between two consecutive points in time.

Quantification of change magnitude is captured by $\xi \subseteq \mathbb{R}$, which we define as the first derivative of ζ, thus expressing the rate of change between two consecutive data points. Namely:

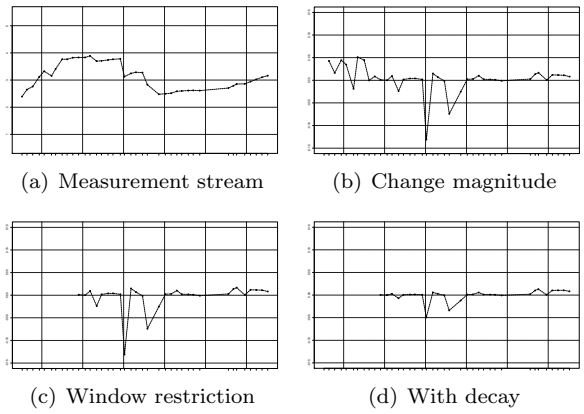
$$\xi = \{\frac{x_i - x_{i-1}}{t_i - t_{i-1}} \mid (x_i, t_i) \in \zeta \text{ and } 1 \leq i \leq n\}$$

Notice that $|\xi| = |\zeta| - 1 = n$, since no change can be defined for the first data point.

It may be desirable to either focus the analysis on a specific time period, or simply limit the number of data points taken into account. For that, instead of considering all the points in the set, one can limit the data points analyzed by using a time window, for example defined by an endpoint k and a width ω. This modifies the formula for ξ to:

$$\xi = \{\frac{x_i - x_{i-1}}{t_i - t_{i-1}} \mid (x_i, t_i) \in \zeta \text{ and } k - \omega \leq i \leq k\}$$

Notice that the window is defined with respect to logical time, but a window based on physical time would also be possible.



(a) Measurement stream (b) Change magnitude

(c) Window restriction (d) With decay

Fig. 3. Impact of the transformations applied to a measurement stream for abrupt change detection.

Since a measurement stream is continuously being updated, one may want to reduce the weight of "older" data points. This puts the focus of the event detection on the recent history by reducing the impact of past trends on the event selection. One can smoothly decrease the importance of past events by using a weighing function, such as a negative exponential. In that case, given a decay factor $\beta \in \mathbb{R}^+$ where $\beta \geq 1$, and with respect to the endpoint of the time window k, a past event at time i (where $i \leq k$) will have a weight $\beta^{i-k} \in \mathbb{R}^+$. This can then be introduced in the formula as:

$$\xi = \{\frac{x_i - x_{i-1}}{t_i - t_{i-1}} \cdot \beta^{i-k} \mid (x_i, t_i) \in \zeta \text{ and } k - \omega \leq i \leq k\}$$

For example, with $\beta = 2$, for the last data point in the window we have $i = k \Rightarrow i - k = 0 \Rightarrow \beta^{i-k} = 2^0 = 1$, for the one but last data point we have $i = k - 1 \Rightarrow i - k = -1 \Rightarrow \beta^{i-k} = 2^{-1} = 0.5$, and so forth. The change magnitudes will thus be multiplied by a weight that continuously decreases with respect to the distance from the end of the window.

The impact of each of the transformations is exemplified in Figure 3. The transformed change magnitude stream can then be used to select events.

The simplest way of selecting events is to define a fixed threshold ι and choosing the data points with an absolute magnitude higher than this value. We can then define the set of abrupt change events as:

$$\Omega_a = \{i \mid |\xi_i| \geq \iota \text{ and } \xi_i \in \xi\}$$

This restricts potential events by a minimum interest level.

Ideally one would like the event selection to adapt to different periods of activity, becoming more sensitive when activity is low, and vice-versa. In case we have a considerable number of data points, we can simply select the events from the tails of the distribution of ξ. We can then create two adaptive thresholds, q_l and q_u as being the first (lower) and last (upper) quantiles of the absolute values of ξ, for a given quantile size α. For example, if $\alpha = 10\%$, q_u is the 9th decile and q_l is the 1st decile, which means that 20% of the data points are potential events. The definition of Ω_a then becomes:

$$\Omega_a = \{i \mid (\xi_i \leq q_l \text{ or } \xi_i > q_u) \text{ and } |\xi_i| \geq \iota \text{ and } \xi_i \in \xi\}$$

Notice that these two thresholds, q_u and q_l, potentially change with each new data point arriving in the stream, thus continuously adapting to the patterns of activity.

B. Sustained change

We define sustained change as a continued increase or decrease throughout a period of time, logical or physical. Quantification of sustained change is captured by $\gamma \subset \mathbb{R}$, which we define recursively as:

$$\gamma_1 = x_1 - x_0$$

$$\gamma_i = \begin{cases} 0 & \text{if } i-1 \in \Omega_a \\ x_i - x_{i-1} & \text{if } sgn(x_i - x_{i-1}) \neq sgn(\gamma_{i-1}) \\ \gamma_{i-1} + (x_i - x_{i-1}) & \text{otherwise} \end{cases}$$

where *sgn* is the *signum* function.

Notice that γ works as an accumulator of the amount of change between consecutive snapshots, restarted when the direction of change shifts. The accumulator also gets reset to 0 when the data point is an abrupt event, because the purpose of analyzing sustained change is to detect situations of slow but steady increase, that would otherwise go under the radar. Furthermore, an abrupt change event almost always induces a sustained change event. In these situations we consider that the former is more important than the latter, *i.e.*, abrupt change overrides sustained change.

Events can be selected based on γ by defining a threshold value τ and choosing the data points where the absolute value of the sustained change is higher than τ. Namely:

$$\Omega_s = \{i \mid |\gamma_i| \geq \tau \text{ and } \gamma_i \in \gamma\}$$

The choice of τ depends on the type of measurement data in the underlying stream and on the desired sensitivity of the detection algorithm in a particular context of use. An example is provided next.

V. Application

The alert service was used for a period of 20 months for a subset of the systems monitored by SIG. The 413 systems in this subset had volumes between 10.000 and 2 million lines of code, with an average of 141.000.

The thresholds that were used during this evaluation period were based on an experiment with two systems we consider typical for systems encountered by SIG. The experiment involved a number of experts manually classifying each alert based on whether the change causing that alert was interesting for them. During this experiment, we found that around 20 percent of the snapshots contain rating changes that we consider noteworthy.

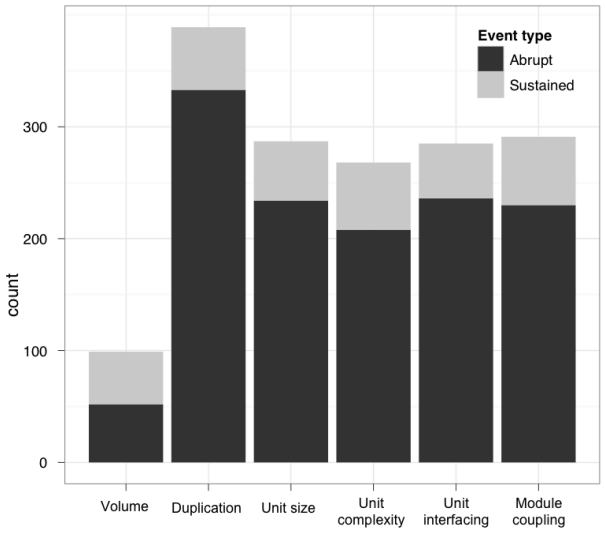


Fig. 4. Number of events per property, grouped by event type.

Period	Dec 2009 - Jul 2011
Monitored systems	413
People notified	43
Snapshots	7240
Snapshots with alerts	876

TABLE I
SUMMARY OF ALERT SERVICE EVALUATION PERIOD.

The following table summarizes the setting for this period:

Out of 43440 (7240 snapshots x 6 system properties) measurement data points generated for the set of systems monitored, approximately 4% (1619) were classified as events, causing 876 alerts (one alert can be caused by multiple events). Figure 4 shows the number of events per property, grouped by event type.

Volume is the property for which less alerts were generated. Furthermore, it is the only property for which more sustained events than abrupt events were detected. This is according to expectations, since:

- For systems under maintenance, volume typically does not change significantly, since the main functionality is already implemented, and changes to the system consist mainly of bug fixes or quality improvements;
- For systems under development, volume will typically grow continuously for long periods while new functionality is being added, thus making sustained change events more likely to occur than abrupt ones.

To determine whether the system was detecting events in a manner corresponding to expert expectations, a systematic evaluation was performed.

VI. Evaluation

In his MSc thesis, Frank Versnel evaluated the accuracy and usefulness of the alert service's event detection mechanism [16]. To achieve this, Versnel identified two alert properties:

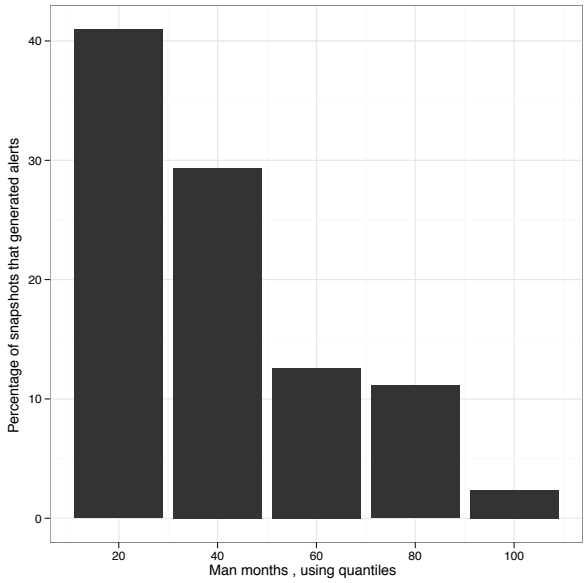


Fig. 5. Percentage of alerts, grouped by system size.

Expected If the change that triggered the alert was expected by the expert monitoring the system.

Desirable If the alert gave the expert more insight in the system's status.

Versnel used a random sample of 98 alerts from the evaluation period described in Section V, and interviewed the receivers of these alerts to determine their desirability and expectancy. He found that 57 out of 98 (58%) of alerts were considered both desirable and unexpected, as depicted in Figure 7.

Apart from determining if alerts were desirable and/or expected, Versnel also asked the experts to categorize the cause of the alert. He identified four categories: *addition*, *modification*, *measurement error*, and *scope error* (the alert was caused by taking code that was not part of the system into account).

Based on this categorization, which is depicted in Figure 6, Versnel found an unexpected use of the alert service. All alerts that were caused by measurement and scoping errors were considered desirable, which indicates that the alert service, apart from events in the system being monitored, was also used to detect errors in the analysis.

Versnel also explored the relation between a size of a system and the chance of a snapshot triggering an alert. He divided the systems from his sample into five quantiles based on system size. The results are depicted in Figure 5.

As can be observed in the chart, smaller systems have a higher alert frequency. This was expected, since most of the aggregation used for calculating the ratings is based on percentages of lines of code, which are easier to change radically the smaller the code base is. However, this does

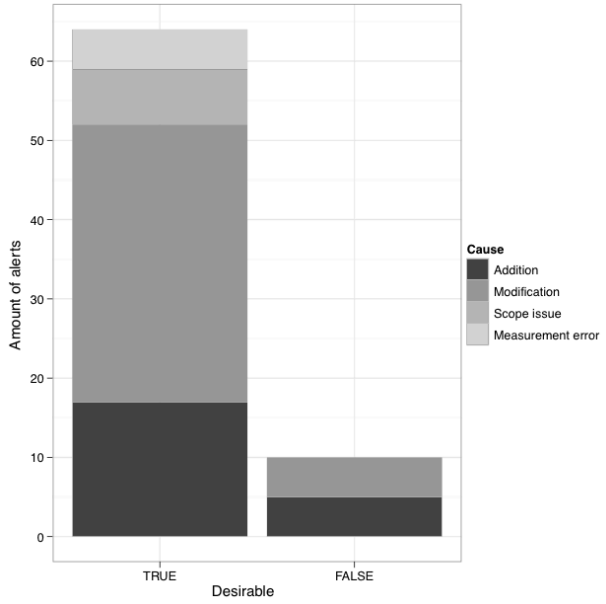


Fig. 6. Categorization of alerts.

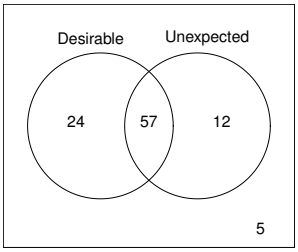


Fig. 7. Desirability and expectancy of alerts.

mean that the alert service is less sensitive for larger systems.

Finally, Versnel looked at the correlation between development activity on a system and the number and desirability of alerts that system generated. He found that systems that are still under development cause a relatively high number of undesirable alerts. One reason for this is that these systems are still quite small, which means that a relatively high percentage of the code is touched when making changes. This causes the alert service to generate alerts that are generally too minor to be interesting. Also, systems under development will generate alerts for volume that are not interesting because they correspond to expectations.

VII. Discussion

The alert service described in this paper has proven to be useful within the context of the Software Improvement Group. We believe that the general idea is also applicable in other situations.

One potential issue with introducing the alert service in a new environment is choosing the thresholds for sustained and abrupt event detection. Once the alert service is operational these thresholds can be changed based on feedback, as was described in Section VI. However, when the alert service is first introduced this information is not yet available.

The initial thresholds mentioned in this paper were based on an experiment with two systems. We changed the thresholds to determine the ideal percentage of snapshots that should trigger an alert. After manual inspection of all alerts that would be generated we found that 20% of snapshots triggering an alert produced the best results. Using a higher percentage would remove some of the false positives, but it would also introduce a large amount of false negatives (events too minor to be interesting). Also, using this approach to determine the initial thresholds has the assumption that the behavior of the two systems is representative for other systems. During the evaluation period, 12% of snapshots triggered an alert, a number reasonably similar to the 20% from the experiment. This indicates that the ideal threshold for different systems is at least comparable.

However, a potential improvement would be to periodically adjust the thresholds depending on feedback. This would eliminate the problem of having two systems that are representative for all systems being monitored, something that might not even be possible if there are many systems with different characteristics. One solution for this would be to use different thresholds for different groups of systems, depending on their size or their development phase.

The approach used to detect sustained events might not be appropriate in a setting where there are large variations in the observed systems' sizes. Sustained events are detected using an absolute threshold, which could make it too sensitive for very small systems. The smallest systems in the evaluation period were around 50.000 lines of code. The reliability of the sustained event detection for smaller systems cannot be guaranteed and needs further research. This problem does not affect the detection for abrupt changes, as that uses the system's change history to set a system-dependent threshold.

Another issue with sustained events is that in some cases the fact that a trend is being followed might not be interesting. During the evaluation period some sustained alerts were considered expected but not desirable, which implies the expert monitoring the system was already aware of the trend that caused the alert. These alerts are not really false positives, since they do represent a noteworthy change in the system. Additional research is therefore needed to decide on how to handle these cases.

VIII. Related work

The work presented in this paper draws influence from the fields of Statistical Process Control (SPC) and data stream management and mining. Related work also exists in the field of time series analysis.

a) Statistical Process Control: SPC is a method for monitoring a production process based on continuous measurement and the application of statistical methods to detect significant deviations that may impact the quality of the final product.

For the original publications on SPC we refer the reader to Shewhart [14], [15].

Applying SPC for software engineering processes has been a topic of some debate (see for example Weller *et al.* [17]) and, although part of the requirements for achieving CMMI levels 4 and 5, it is far from being an established practice. Baldassare *et al.* [4] presented another attempt at clarifying and better defining how SPC can be employed for monitoring software processes.

Our approach differs from traditional SPC and its existing applications to software engineering in that we introduce automation and perform monitoring of the *product* rather than the process.

b) Data stream management and mining: A *data stream* is a continuous ordered sequence of incoming data generated by a certain process or activity. Examples include computer network traffic, ATM transactions, sensor data, etc. The field of data stream management and mining is concerned with how to handle large quantities of data from such streams and how to efficiently use them to automatically gather knowledge about the underlying process or activity. For more information we refer the reader to Gaber *et al.* [8] and Golab *et al.* [9].

Fawcett & Provost [6] define a framework for *activity monitoring* which they define as "monitoring the behaviour of a large population of entities for interesting events requiring action". Their work is concerned with providing a platform for formalizing this type of problems, and not providing a specific method for detecting events in a certain context.

c) Time series analysis: Another paradigm for analyzing sequences of temporally ordered data points is that of *time series analysis*. It comprises methods for extracting statistics and characteristics from time series data and it is used in the areas of signal processing and mathematical finance.

An example of its usage for detection of outliers and change points in the data is that by Yamanishi *et al.* [18]. Another example of its application is Hindle *et al.* [11], who used Fourier analysis to uncover natural periodicities from the logs of change events.

Time series analysis typically relies on the availability of a large amount of data and on some statistical assumptions of the underlying process. From our observations, these assumptions do not hold for software measurement streams. Furthermore, the amount of data available is typically of a lower order of magnitude.

IX. Conclusions and future work

d) Contributions: In this paper, we presented our approach for automatic event detection in source code measurement streams, generated in the context of software product monitoring. Our method is inspired by data stream processing and Statistical Process Control, but is specifically tailored to take into account the characteristics of source code metrics. These metrics produce a considerable amount of data, which is too large to manually inspect every measurement to determine whether it is noteworthy. We provided a formal treatment of the problem and our solutions for detecting two types of events: *abrupt change* and *sustained change*.

We described the application of the method in an industrial setting where the detection method is used as part of an alert service that warns software engineering experts about significant events in software products being monitored.

We evaluated this case study quantitatively to establish the effectiveness of our methodology to achieve rapid feedback and high scalability.

e) Future work: The next steps will be mainly to focus on a more thorough validation of the method. These include quantitative evaluations of:

- the impact of each of the components of the event detection algorithm;
- the impact of changes in the particular thresholds of each of these components;
- the performance of the alert service in terms of false negatives.

We also believe that an implementation of the alert service for open source projects would be valuable for the community. Some of these projects have a large number of contributors, and not every contributor might have time to review every commit. Using the alert service would provide a mechanism to inform about noteworthy changes to the project's codebase.

A similar approach can be taken in projects where the development team is spread over multiple locations. In this case the alert service could be used as an additional communication tool, making it more transparent what areas of the system the different teams are working on.

References

[1] T. L. Alves, J. P. Correia, and J. Visser. Benchmark-based aggregation of metrics to ratings. In K. Matsuda, K. ichi Matsumoto, and A. Monden, editors, *IWSM/Mensura*, pages 20–29. IEEE, 2011.

[2] T. L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *ICSM*, pages 1–10. IEEE Computer Society, 2010.

[3] R. Baggen, J. P. Correia, K. Schill, and J. Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, pages 1–21, May 2011.

[4] M. T. Baldassarre, N. Boffoli, G. Bruno, and D. Caivano. Statistically based process monitoring: Lessons from the trench. In Q. Wang, V. Garousi, R. J. Madachy, and D. Pfahl, editors, *ICSP*, volume 5543 of *Lecture Notes in Computer Science*, pages 11–23. Springer, 2009.

[5] E. Bouwers and R. Vis. Multidimensional software monitoring applied to erp. *Electr. Notes Theor. Comput. Sci.*, 233:161–173, 2009.

[6] T. Fawcett and F. Provost. Activity monitoring: noticing interesting changes in behavior. In *KDD '99: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 53–62, New York, NY, USA, 1999. ACM.

[7] N. Fenton and S. L. Pfleeger. *Software metrics (2nd ed.): a rigorous and practical approach*. PWS Publishing Co., Boston, MA, USA, 1997.

[8] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: a review. *SIGMOD Rec.*, 34(2):18–26, 2005.

[9] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[10] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In R. J. Machado, F. B. e Abreu, and P. R. da Cunha, editors, *QUATIC*, pages 30–39. IEEE Computer Society, 2007.

[11] A. Hindle, M. W. Godfrey, and R. C. Holt. Mining recurrent activities: Fourier analysis of change events. In *ICSE Companion*, pages 295–298. IEEE, 2009.

[12] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[13] T. Kuipers and J. Visser. A tool-based methodology for software portfolio monitoring. In M. Piattini and M. A. Serrano, editors, *Software Audit and Metrics*, pages 118–128. INSTICC Press, 2004.

[14] W. A. Shewhart. *Economic control of quality of manufactured product / by W. A. Shewhart*. London„ 1931.

[15] W. A. Shewhart. *Statistical method from the viewpoint of quality control*. The Graduate School, Washington, 1939.

[16] F. Versnel. Evaluation and improvement of software monitor alerts. Master's thesis, University of Amsterdam, 2010.

[17] E. F. Weller, D. Card, B. Curtis, and B. Raczynski. Point/counterpoint. *IEEE Software*, 25(3):48–51, 2008.

[18] K. Yamanishi and J. ichi Takeuchi. A unifying framework for detecting outliers and change points from non-stationary time series data. In *KDD*, pages 676–681. ACM, 2002.