Probing methods for Automatic Error Resolution in a heterogeneous software environment

Antonio Pierro[*], Salvatore Di Guida[†], Vincenzo Innocente[†] and Ilja Kuzborskij [**]

[*]*INFN-Bari - Bari University, Via Orabona 4, Bari 70126, Italy*
[†]*CERN Geneva 23, CH-1211, Switzerland*
[**]*Vilnius University, 3 Universiteto St, LT-01513 Vilnius, Lithuania*

Abstract.
We investigate the feasibility to improve the error resolution through automation like natural language-based and statistical analysis algorithms, in order to detect errors and security issues, and to convert error messages from encrypted into human-readable ones in a heterogonous software environment. To reach this goal we study a real case using the data extracted from PopCon, a package used for the population of CMS Condition Databases, that is embedded into CMS Software framework, CMSSW, and relies on different underlying applications such as ORACLE, POOL, CORAL in order to perform database transactions.

Keywords: CMSSW, CORAL, POOL, error detection, google, natural language processing
PACS: 89.20.Ff

INTRODUCTION

In general, the time and energy that physics, engineers and informatics working on both LHC[1] experiments and Grid services spend on documenting, recreating and attempting to resolve software bugs and reported issues (i.e. "application problems") can be very large.

For this reason, in this paper we try to study the technical feasibility to use semi-automatic detection of error, and give to the final user a more readable log, thanks to the support of developers of each software subsystem and the feedback of other final users. We study this approach in a real context and discusses the benefits, the potential pitfalls.

Problem Definition

Before we proceed to problem analysis and solution estimation, let us describe problem cases we have encountered, while working in the context of CMSSW[2] framework.

First of all we would like to draw your attention to first class of error-generating entities, which basically are major software packages. We do not feel that description of minor packages or programs is really needed in this case, since it is very common situation, when error from such source is being propagated to higher abstraction tiers. Main software units include:

CMSSW framework - the core system; PopCon[3] tool - CMSSW subsystem; POOL & CORAL - external packages, which CMSSW and its subsystems extensively rely on Oracle DBMS - fundamental database management system of CMSSW.

Second class of error-generating entities are implicit and does not exist in form of software of or another unit. It is a metadata-dependent error class which can be detected only by observation of specific ”points” from time to time. We call this approach time inconsistency check. These checks can point out such errors like hardware malfunction or unknown, undetected software failure.

It is very important in out case, to distinguish properties and behavior of every error source in order to perform complete problem analysis, that is why it is crucial to detail life cycle of error in the context of software entities and their environment.

- *Errors by scripts of CMSSW framework.*
  From the error generation point of view, the most important feature of CMSSW framework is processing of

python scripts. This task is accomplished by cmsRun module. The framework itself enables a way to log all the jobs run by it using a module called Framework Job Report. This module can be called either inside the script, or explicitly using an option of cmsRun (namely,”-j <filename>.xml”). Framework Job Report yields a XML file with all informati on about an error, labeled by the tag ”FrameworkError”, an exit status (a number) and the type of error. Besides, an accompanying string with the error output is provided.

- *Error by executables built inside the framework.*
  Some binaries built inside the framework make use of the PopCon, POOL and CORAL libraries. The error-status of such applications can be monitored using log files (a part of them is already stored into the POPCONLOG database account) to be parsed every time the application runs. The error messages, indeed, can be modified when a new version of the software is released: usually, an improvement of the software leads to clearer error messages, but, in some cases, we can expect that the error message becomes less readable.
- *Time inconsistencies.*
  Usually, a time inconsistency, in our use cases, is related to a hardware failure (network failure or machine powered off). This can be already detected using the DB back-end provided by so called ”log file tail fetcher”, the tool, which parses log files and puts timestamp data into the log database. Here a comparison between two timestamps is now possible. Another example of time inconsistency metadata exportation is quota checking script. The prime goal of quota checking script is to record information about account quota state. The recording process is similar to ”log file tail fetcher”. But on the other hand quota checking maintains another feature - it stores along timestamp of exportation. This gives a possibility to perform time inconsistency test. In general case all automatic exportation tools or automatic DB populators of should include this feature, so the inconsistency test becomes possible.

Solution & Goals

Since problem is defined, we can study the goals. The basic solution vision covers all problematic cases mentioned above and intended mainly for user-side features. Although, some developer features and bonuses will be also taken into account.

- *User-friendly error messages*
  Among the most important use-cases that could possibly be achieved by error detection system there is one, that gives user some information about occurred error. Of course this is hardly possible without careful error detection system maintenance, which can have semi-automated features, like, data mining. As a good option some automatic notification to developers or administrators can be provided in situation when error detection is helpless. For instance, if new, unknown yet error occurs, system can queue it for resolution by maintainers.
- *Application to heterogeneous software environment*
  Being a part of larger system, PopCon heavily relies on infrastructure software packages as was mentioned before in problem definition section. Because of this close integration it is inefficient to monitor erroneous situations only in a distinct environment like PopCon - errors are often propagated from different sources. Of course we could expand error detection scope to some extent, but this would proof unreliable practice, since some packages are constantly changing and modules to these packages are written by different developers, who are very often unconstrained by some error format standard. As a common solution, population of error detection related database can be accomplished thanks to general “Wiki” documentation approach used by CMS developers.
  All in all, ability to detect and reason errors from different software sources would be very important property of detection system. This brings resilience in terms of version and software independence to some point.
- *Inconsistency notification*
  Here, we address, generally speaking, second class of error-generating entities, in other words inconsistencies - in more specific, time inconsistencies. Time inconsistencies are important in the sense of error prediction. Speaking of relationships between inconsistencies, time inconsistencies are basically tied to data inconsistencies, because consistency markers (e.g. timestamps) are commonly placed by data importers. That is why detection of time inconsistency can tell, for instance, about hardware failure and as a result about failed data transaction.
  This type of error can be a warning sign for a user, notifying about some kind of possible data inconsistency or system instability. On the other hand, this is major error note for a developer or administrator, who maintains the system.

SOLUTION METHOD OVERVIEW

Though we studied most of functional and non-functional requirements for this experimental system, we are going describe feasibility of only not relatively easily implemented use-case - natural language processing. Other ones, for example XML parsing, are not really worth specific interest, since there are many well-working stabile methods for realizing these requirements. On the other hand natural language processing is not widely used. That is why, we spent particular attention to this problem in next section.

VALIDATION OF NATURAL LANGUAGE-BASED SIMILARITY CHECK

In this section we describe how we validate and tune algorithms and methods in order to find similar sentences across multiple text-log files produced by several applications using PopCon.

First of all, we make a research concerning open source projects facing the issue of measuring the semantic similarity of texts.

The best method that meets our requirements is an experimental module, which uses algorithm for sentence similarity detection described in [4] using three different methods: *string similarity*, *semantic similarity* and *word order similarity*.

The *string similarity* method pays attention to string-similarity checking, which is important in the case of specific domain-related tags, e.g. Oracle error messages, which are not available in word dictionaries or CMSSW signatures, which are unavailable in both dictionaries and corpora.

The *semantic similarity* method gives good results on matching fuzzy, but sense-close sentences, which is the case of Versionable error resolution.

The *word order similarity* method provides information about the relationship between words: which words appear in the sentence, and which words come before or after other words. Both of these semantic and syntactic (in terms of word order) pieces of information play a role in comprehending the meaning of sentences in versionable errors.

Unfortunately, the Corpus-based method[1] used for experimental runs in the paper cited above has several drawbacks for our purposes. It is not available free of charge and it is not suitable for production purposes at our scale; hundreds of plain text-log grouped in different sentences, each of them reflects the exception propagation raised by an error along the stack and each of plain text error can contains over 100 words. Hence, these issues give limitations in terms of performance because it requires precompilation; besides we didn't success to get similar sentences since the original Corpus is suitable for natural language processing and it is not correlated with our field of interest.

Therefore, we reused *Google Similarity Distance*[5] web-search engine based approach and *NLTK*[6] (Natural Language Toolkit) python package WordNet to customize our algorithm.

The former, *Google Similarity Distance* web-search engine, was used to substitute semantic similarity in order to take profit by the ability of *Google's Crawl* to parse all CERN pages, LaTex sources and "public" development forums like *Oracle-DBA-OCP* forum and by doing this parse we get Technical Knowledge Base information for our problem domain. The latter, *NLTK* python package uses WordNet to augment semantic word similarity estimation. In more details, for each comparison of two sentences, (s_1, s_2), the algorithm yelds a "number of similarity" ranging between 0 and 1; this number will be called "estimation value"; in mathematical form: we define a function E, on the cartesian product of the sets of error sentences A_n and A_m, where n and m, with $n \neq m$, are tags for different version of the software.

$$E : (s_i^{(n)}, s_j^{(m)}) \in A_n \times A_m \longmapsto e = E(s_i^{(n)}, s_j^{(m)}) \in [0,1]$$

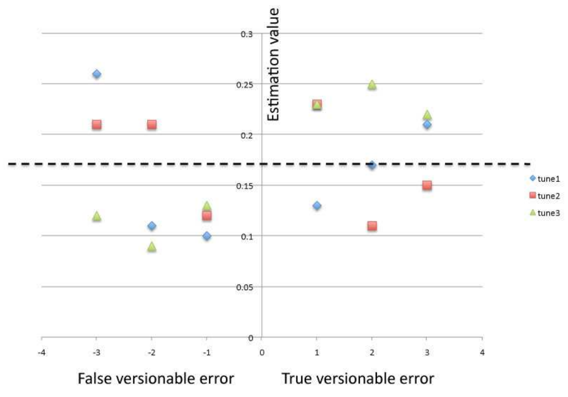
Since the result of algorithm depends by the contribution of the following methods: the *Semantic Similarity Methods* in *WordNet* like WordNet::Similarity::path, and *Google Distance*, we decided to tune the main algorithm changing the weight of contribution of these methods. In mathematical form the output result was tuned weighting the methods' contribution:

$$E = \sum_{k=1}^{n} w_k * m_k \quad w_k \in [0,1] \quad \sum_{k=1}^{n} w_k = 1$$

[1] Corpus-based method is the study of language as expressed in samples (corpora) or "real world" text.

where

$$m_k \in [GoogleDistance, WordNet :: Similarity :: method_1, \ldots, WordNet :: Similarity :: method_{n-1}]$$



1: Graphical model validation used to tune the algorithm for measuring the semantic similarity

In order to establish the validity of the tuning for tracking straight similar sentence, we report the "estimation value" result in a plot divided in two parts; in the former, on the negative x-axis, you can find the comparison between two sentences which are not Versionable errors (namely, false Versionable errors), while the positive x-axis contains the so-called true Versionable errors, i.e. comparisons between sentences reporting a Versionable error; on the y-axis, finally, we put the values obtained from the estimation using the algorithm described above. At a first glance, if the algorithm is well tuned, we expect that the y-axis and a straight line parallel to the x-axis should divide the diagram into four parts. One quadrant contains the estiamtion value for all the true Versionable errors and the opposite one the values belonging to false Versionable errors. Instead, in case of "bad tune" (i.e., the algorithm is not able to distinguish true and false versionable error), is not possible to find a straight line parallel to x-axis dividing true Versionable errors and false Versionable error sinto two separate sets. The final result of our test, during the validation of tuning of weight of methods can be summarized in the following plot. As you can see, only one tuning (namely number 3) suits our procedure to find the validity of the algorithm. The tune suiting the similarity sentence successfully was obtained assigning a low-weight to *Google similarity distance* method (about 0.2) and an high-weight to the WordNet::Similarity::path method (about 0.8).

CONCLUSION

In this paper we focused on the possibility to use methods for measuring the semantic similarity in order to find Versionable error between error sentences. The result demonstrates that the method is feasible since the application chosen, PopCon, is well integrated in the contet of a large experiment and of a heterogeneous software environment. Hence, we are optimistic on the possibility to apply this work to other software packages having the problem of Versionable error. This kind of error is very common, since, in order to meet the new requirements for one or more LHC experiments, the software is continuously updated and developed, and new versions of software are made available to the end-users. Therefore, having the possibility to automate the identification of Versionable errors will allow to save time during the crucial and decisive phase in prevision of the LHC start-up.

REFERENCES

1. The LHC Project. LHC Design Report, Volume I: the LHC Main Ring. Technical Report CERN-2004-003-V-1, CERN, Geneva, 2004. URL
2. CMS Computing TDR, CERN-LHCC-2005-023, <http://cdsweb.cern.ch/record/838359> 20 June 2005.
3. PopCon (Populator of Condition Objects). First experience in operating the population of the “condition database” for the CMS experiment. International Conference on Computing in High Energy and Nuclear Physics, March 2009
4. Semantic Text Similarity using Corpus-Based Word Similarity and String Similarity; ACM Transactions on Knowledge Discovery from Data (TKDD) (2008)
5. Rudi L. Cilibrasi and Paul M.B. Vitanyi - The Google Similarity Distance, 30 May 2007
6. Sean Boisen, SemanticBible - Natural Language Processing in Python using NLTK, LinuxFest Northwest 2008