A Framework for the Discovery of Predictive Fix-time Models

Francesco Folino, Massimo Guarascio and Luigi Pontieri
Institute ICAR, National Research Council (CNR), via P. Bucci 41C, 87036 Rende, CS, Italy

Keywords: Data Mining, Prediction, Business Process Analysis, Bug Tracking.

Abstract: Fix-time prediction is a key task in bug tracking systems, which has been recently faced through the definition of inductive learning methods, trained to estimate the time needed to solve a case at the moment when it is reported. And yet, the actions performed on a bug along its life can help refine the prediction of its (remaining) fix time, possibly with the help of Process Mining techniques. However, typical bug-tracking systems lack any task-oriented description of the resolution process, and store fine-grain records, just capturing bug attributes' updates. Moreover, no general approach has been proposed to support the definition of derived data, which can help improve considerably fix-time predictions. A new methodological framework for the analysis of bug repositories is presented here, along with an associated toolkit, leveraging two kinds of tools: *(i)* a combination of modular and flexible data-transformation mechanisms, for producing an enhanced process-oriented view of log data, and *(ii)* a series of ad-hoc induction techniques, for extracting a prediction model out of such a view. Preliminary results on the bug repository of a real project confirm the validity of our proposal and, in particular, of our log transformation methods.

1 INTRODUCTION

In general, issue tracking systems (a.k.a. "trouble/incident ticket" systems) are commonly used in real collaboration environments in order to manage, maintain and help resolve various issues in an organization/community. A popular sub-class of these systems it that of bug tracking systems, aimed at supporting the fixing of bugs in software artifacts, and widely used in complex software-development projects, especially in the open-source world.

A key task in such a context amounts to accurately foreseeing a bug fix time (i.e. the time needed to eventually solve the bug). This problem recently attracted the attention of data-mining researchers (Anbalagan and Vouk, 2009; Marks et al., 2011; Panjer, 2007), who tried to extract either a discrete (i.e. classification-oriented) or continuous (i.e. regression-oriented) fix-time predictor, out of historical bug logs. Current solutions rely on standard propositional prediction methods, while regarding each bug record as a tuple encoding all information available when the bug was initially reported, and labelled with a discrete or numerical (target) fix-time value. In this way, the rich amount of log data collected across the life of each bug — including any change made to bug properties, like its priority, criticality, status, or assignee — is disregarded, despite it may well help update, at run-time, the prediction of (remaining) fix times.

The analysis of activity logs is the general aim of Process Mining research (van der Aalst et al., 2003), which recently started facing right the induction of predictive process models (van der Aalst et al., 2011; Folino et al., 2012; Folino et al., 2013). However, these approaches need a mapping of log records to well-specified process tasks, which are rarely defined in real systems, where the logs typically register only the sequence of changes made to a bug's attributes. In fact, despite many systems support the design of bug-handling workflows, these are rarely used in real applications. Moreover, different bug repositories tend to exhibit heterogeneous data schemes (even if built with the same system, such as, e.g., Bugzilla), by virtue of the possibility, offered by most tracking platforms, to customize the data fields of bugs.

In this work, we propose a comprehensive methodological framework for the analysis of bug data and, in particular, for the discovery of fix times, which allows for taking full advantage of bug attributes and bug modification records, so overcoming the limitations of current solutions. In particular, in order to help the analyst grasp a suitable abstraction level over bug histories, we define a modular set of parametric data-transformation methods for converting each bug history into a process trace (where update records are abstracted into higher-level activ-

Folino F., Guarascio M. and Pontieri L..
A Framework for the Discovery of Predictive Fix-time Models.
DOI: 10.5220/0004897400990108
In *Proceedings of the 16th International Conference on Enterprise Information Systems* (ICEIS-2014), pages 99-108
ISBN: 978-989-758-027-7

ities), and for possibly enriching these traces with derived/aggregated data. In this way, a high-quality process-oriented view of bug histories can be obtained and analyzed with existing (or novel) process mining methods, in order to eventually build a predictive model, capable to estimate, at run-time, the remaining fix time of a bug. The approach has been implemented in a system prototype, offering an integrated and extensible set of data-transformation and predictive learning tools.

By virtue of its generality and flexibility, the proposed approach can be applied profitably to a variety of real-life bug repositories, while allowing the analyst to customize the discovery of a fix-time model to the specific data schema and business rules of the repository under analysis. Moreover, as the approach only assumes that each log event represents a modification to a case attribute, it can be easily extended to analyze the logs of other lowly-structured process management systems (such as, e.g., issue-tracking systems or data-centric transactional systems).

The rest of the paper is structured as follows. Section 2 summarizes some relevant related works, and the main points of novelty of our proposal. After introducing a few basic concepts in Section 3, we illustrate, in Section 4, our core log-abstraction methods. The overall discovery approach and the implemented system are presented in Sections 5 and 6, respectively. We then discuss a series of tests in Section 7, and draw some concluding remarks in Section 8.

2 RELATED WORK

Previous approaches to the forecasting of bug fix times mainly rely on the application of classical learning methods, devised for analysing propositional data labelled with a discrete or numerical target. In particular, linear regressors and random-forest classifiers were trained in (Anbalagan and Vouk, 2009) and in (Marks et al., 2011), respectively, in order to predict bug lifetimes, using different bug attributes as input variables. Different standard classification algorithms were exploited instead in (Panjer, 2007) to the same purpose. Decision trees were also exploited in (Giger et al., 2010) to estimate how promptly a new bug report will receive attention. Moreover, a standard linear regression method was used in (Hooimeijer and Weimer, 2007) to predict whether a bug report will be triaged within a given amount of time.

As mentioned above, none of these approaches explored the possibly to improve such a preliminary estimate subsequently, as long as the bug undergoes different treatments and modifications. The only (partial) exception is the work in (Panjer, 2007), where some information gathered after the creation of a bug is used as well, but just for the special case of unconfirmed bugs, and up to the moment of their acceptation. On the contrary, we want to exploit the rich amount of log data stored for the bugs (across their entire life), in order to build a history-aware prediction model, providing accurate run-time forecasts for the remaining fix time of new (unfinished) bug cases.

Predicting processing times is the goal of an emerging research stream in the field of Process Mining, which specifically addresses the induction of state-aware performance model out of historical log traces. In particular, the discovery of an annotated finite-state model (AFSM) was proposed in (van der Aalst et al., 2011), where the states correspond to abstract representations of log traces, and store processing-time estimates. This learning approach was combined in (Folino et al., 2012; Folino et al., 2013) with a predictive clustering scheme, where the initial data values of each log trace are used as descriptive features for the clustering, and its associated processing times as target features. By reusing existing induction methods, each discovered cluster is then equipped with a distinct prediction model — precisely, an AFSM in (Folino et al., 2012), and classical regression models in (Folino et al., 2013).

Unfortunately, these Process Mining techniques rely on a process-oriented representation of system logs, where each event refers to a well-specified task; conversely, common bug tracking systems just register bug attribute updates, with no link to resolution tasks. To overcome this limitation, we try to help the analyst extract high-level activities out of bug history records, by providing her/him with a collection of data transformation methods, tailored to fine-grain attribute-update records, like those stored in bug logs.

The capability of derived data to improve fix-time predictions was pointed out in (Bhattacharya and Neamtiu, 2011), where a few summary statistics and derived properties were computed for certain Bugzilla repositories, in a pre-processing phase. We attempt to generalize such an approach, by devising an extensible set of data transformation and data aggregation/abstraction mechanisms, allowing to extract and evaluate such derived features for a generic bug log.

3 PRELIMINARIES

In order to make the discourse concrete, let us focus on the structure of a bug repository developed with *Bugzilla* (`http://www.bugzilla.org`), a general-purpose bug-tracking platform, devoted to support

people in various bug-related tasks – e.g., keep track of bugs, communicate with colleagues, submit/review patches, and manage quality assurance (QA). Notice, however, that this particular choice does not undermine the generality of our approach, since very similar bug tracking strategies take place in most real-life software development/maintenance environments.

In typical Bugzilla applications, tasks are often carried out in an unstructured manner, without being enforced by a prescriptive process model. Such applications mainly looks like a repository, where an extensible set of attributes are associated with a bug, and possibly updated along its entire life.

For example, in the Bugzilla repository of project *Eclipse* (used in our experiments), the main attributes associated with each bug *b* are: who entered *b* into the system (`reporter`); the last solver *b* was assigned to (`assignee`); the `component` and `product` affected by *b*; `severity`'s and `priority`'s levels; the list of users that must be kept informed on *b*'s progress (`CC`); the lists of other bugs depending on *b* (`dependsOn`), and of related documents (`seeAlso`); a `milestone`; the `status` and `resolution` of *b* (both described below).

Few bug attributes (e.g., `reporter`) are static, whereas the others (e.g., `status`, `resolution`, `assignee`) may change as long as the bug case evolves. In particular, the `status` of a bug *b* may take the following values: *unconfirmed* (i.e. *b* was reported by an external user, and it needs to be confirmed by a project member), *new* (i.e. *b* was opened/confirmed by a project member), *assigned* (i.e. *b* was assigned to a solver), *resolved* (i.e. a fix was made to *b*, but it needs to be validated), *verified* (i.e. a QA manager has validated the fix), *reopened* (if the last fix was judged incorrect), and *closed*. For a resolved bug *b*, the `resolution` field may take one of these values: *fixed*, *duplicate* (i.e. *b* is a duplicate of another bug), *works-for-me* (i.e. *b* has been judged unfounded), *invalid*, *won't-fix*.

In any Bugzilla repository, the whole history of a bug is stored as a list of update records, all of which share the same structure, consisting of five predefined fields (in addition to a bug identifier): *who* (the person who made the update), *when* (a timestamp for the record), *what* (the attribute modified), *removed* (the former value of that attribute) and *added* (the newly assigned value). Figure 1 reports, as an example, the update records of an Ecplise's bug.

Bug Traces and Associated Data: The contents of a bug repository can be viewed as a set of bug *traces*, each storing the sequence of *events* recorded during the life of the bug. As explained above, each of these events concerns the modification of a bug attribute,

Who	When	What	Removed	Added
svihovec	2012-06-20 10:12:27 EDT	CC		margolis, svihovec
		Target Milestone	---	0.8.1 Final
jinfahua	2012-06-20 22:21:22 EDT	CC		jinfahua, pfyu
		Assignee	edt.ide.ui-inbox	songfan
pfyu	2012-06-21 03:52:44 EDT	Attachment #217671 Flags		review?
svihovec	2012-06-21 11:09:34 EDT	CC		jspadea, jvincens
pfyu	2012-06-25 05:04:17 EDT	CC		wxwu
jspadea	2012-07-02 10:26:15 EDT	Assignee	songfan	jspadea
jspadea	2012-07-02 15:21:21 EDT	Status	NEW	RESOLVED
		Resolution	---	FIXED
lasher	2012-07-18 15:27:24 EDT	Attachment #218182 Flags		iplog+
mheitz	2013-01-03 10:44:55 EST	Status	RESOLVED	CLOSED

Figure 1: Activity log for a single Bugzilla's bug (whose ID is omitted for brevity). Row groups gather "simultaneous" update records (sharing the same timestamp and executor).

and takes the form of the records in Figure 1.

Let E be the universe of all possible bug events, and $\mathcal{T}$ be the universe of all possible bug traces. For any event $e \in E$, let $who(e)$ and $when(e)$, $what(e)$, $removed(e)$, and $added(e)$ be the executor, the timestamp, the attribute modified, the former value and new value stored in e, respectively.

For each (bug) trace $\tau \in \mathcal{T}$, let $len(\tau)$ be the number of events stored in τ; moreover, for any $i = 1 \,..\, len(\tau)$, let $\tau[i]$ be the i-th event of τ, and $\tau(i] \in \mathcal{T}$ be the *prefix* trace gathering the first i events in τ.

Clearly, prefix traces have the same form as fully unfolded traces (and yet belong to $\mathcal{T}$), but only represent partial bug histories. In actual fact, the prefix traces of any bug allow us to look at the evolution of that bug, across its whole life. For example, the activity log of Figure 1 (which just stores the history of one bug) will be represented as a trace τ_{ex} consisting of 12 events, one for each of the update records (i.e. rows of the table) in the figure; in particular, for the first event, it is $who(\tau_{ex}[1]) = svihovec$, $what(\tau_{ex}[1]) = \texttt{CC}$, $added(\tau_{ex}[1])) = \{margolis, svihovec\}$.

As mentioned above, typical bug tracking systems store several attributes for each bug instance (e.g., `reporter`, `priority`, etc.), which may take different values during its life. Let $F_1, \ldots, F_n$ be all of the attributes defined for a bug. Then, for any (either partial of completed) bug trace τ, let $data(\tau)$ be a tuple storing the updated values of these attributes associated with τ (i.e. the values taken by the corresponding bug after the last event of τ), and $data(\tau)[F_i]$ be the value taken by F_i (for $i = 1..n$). Clearly, for any fully unfolded bug trace τ, the data tuple of each sub-trace $\tau(i]$ is a snapshot of the data associated with the bug at the i-th step of its history (with $i \in \{1, \ldots, len(\tau)\}$).

Finally, a (bug) *log* L is a finite subset of $\mathcal{T}$, while the *prefix set* of L, denoted by $\mathcal{P}(L)$, is the set of all possible prefix traces that can be extracted from L.

Fix-time Measurements and Models: Let $\hat{\mu}_F : \mathcal{T} \to \mathbb{R}$ be an unknown function assigning a fix-time

value to any bug (sub-)trace. The value of $\hat{\mu}_F$ is clearly known over all $\mathcal{P}(L)$'s traces, for any given log L — indeed, for any log trace τ and prefix $\tau(i]$, it is $\hat{\mu}_F(\tau(i]) = when(\tau[len(\tau)]) - when(\tau[i])$. For example, for the trace $\tau_{ex}(2]$, encoding the first 2 events in Figure 1, it is $\hat{\mu}_F(\tau_{ex}(2]) = when(\tau[12]) - when(\tau[2])$ = 197 days (assuming that time spans are measured in days).

A *Fix-time Prediction Model (FTPM)* is a model approximating $\hat{\mu}$, which can estimate the remaining fix time of a bug, based on its current trace. Learning such a model is an inductive problem, where the training set is a log L, and the value $\hat{\mu}_F(\tau)$ of the target measure is known for each (prefix) trace $\tau \in \mathcal{P}(L)$.

4 CORE BUG TRACE ABSTRACTION OPERATORS

In the discovery of an *FTPM* model we want to take into account all bugs' histories (i.e. all sequences of update records), in addition to the intrinsic features of the bugs (e.g., the affected product, severity level, reporter). Our core idea is to regard some of the actions performed on a bug as a clue for the activities of an unknown (bug resolution) process, in order to possibly exploit Process Mining approaches. To this end, we discard the naïve idea of just defining such activities as all possible changes to the status of a bug, since this will lead to discard relevant events, such as the (re-)assignment of the bug to a solver, or the modification of key properties (like its severity, criticality, or category). On the other hand, we do not either adopt the extreme solution of looking at all attribute updates as resolution tasks, seeing as many of them are hardly linked to fix times, and they may even have a noise-like effect on the discovery of fix-time predictors.

The rest of this section presents a collection of parametric data-transformation methods, which are meant to turn bug histories into abstract traces of relevant resolution activities, suitable for the application of process-oriented prediction techniques.

Activity-oriented Event Abstraction: An event abstraction function α is a function mapping each event $e \in E$ to an abstract representation $\alpha(e)$, which captures relevant facets of the action performed. To this end, in current process mining approaches, log events are usually abstracted into their associated tasks, possibly combined with other properties of them (e.g., their executors), under the assumption that the events correspond to the execution of work-items, according to a workflow-oriented view of the process analyzed.

In our framework, such a function α is right intended to turn each bug-tracking event into a high-level bug-resolution activity, by mapping the former to a label that captures well its meaning. As a bug system only tracks attribute-update events, the analyst is allowed to define this function in terms of their fields (i.e. *who*, *when*, *what*, *added*, and *removed*).

The default instantiation of α, denoted by $\overline{\alpha}$, is defined as follows (with symbol + denoting the string concatenation operator):

$$\overline{\alpha}(e) = \begin{cases} what(e) + \text{":="} + added(e), \text{ if } what(e) \in \\ \qquad\qquad \{\texttt{status}, \texttt{resolution}\} \\ \text{"}\Delta\text{"} + what(e), \text{ otherwise} \end{cases} \quad (1)$$

This particular definition of α focuses on what bug attribute has been modified, while abstracting any other event's field (namely, *who*, *when*, *removed*, and *added*); as an exception, the assigned values are included in the abstract representation when the update involves the `status` or `resolution`, since such information can help characterize the current state of a bug, and improve fix-time predictions. For example, for the first two events of the bug trace τ_{ex} (gathering all the records in Figure 1), it is $\overline{\alpha}(\tau_{ex}[1]) = \Delta\texttt{CC}$, and $\overline{\alpha}(\tau_{ex}[2]) = \Delta\texttt{TargetMilestone}$, while the activity label of the last event ($\tau_{ex}[12]$) is `status`:=*closed*.

Different event abstraction functions can be defined by the analyst, in order to focus on other facets of bug activities, or to change the level of detail, depending on the specific bug attributes (and associated domains) available in the application scenario at hand. For instance, with regard to the scenario of Section 3, one may refine the representation of severity-level changes by defining two distinct activity labels for them, say Δ*Severity-Eclipse* and Δ*Severity-NotEclipse*, based on the presence of substring "eclipse" in the e-mail address of the person who made the change.

Macro-event Criterion: In real bug tracking environments, multiple fields of a bug are often modified in a single access session, and the corresponding activity records are all stored with the same timestamp, in an almost arbitrary order. For example, in our experimentation, we encountered many cases where the closure of the bug (i.e. an event of type `status`:=*closed*) preceded a "contemporaneous" change of assignee (or a message dispatch).

Regarding each set of contemporaneous events as one *macro-event*, the analyst can define three kinds of data-manipulation rules, in order to rearrange them based on their fields: *(i)* a *predominance* rule, assigning different relevance levels to simultaneous events (with the ultimate aim of purging off less relevant

Table 1: Default macro-event criterion: predominance, merging and sort rules over simultaneous events. No merging rules for levels 2 and 3 (whose events are only reordered), and no sort rules for level-1 events (which are merged together) are defined.

Predominance levels		**Merging rules**	**Sort rules**
(lev.)	(bug attributes)	(macro-event activity label)	(ordering relation)
1	`status, resolution`	$\alpha(\langle \_, \_, \texttt{status}, \_, \_\rangle) + \text{“+”} + \alpha(\langle \_, \_, \texttt{resolution}, \_, \_\rangle)$	—
2	`priority, severity, assignee`	—	`priority`<`severity`<`assignee`
3	`milestone, CC`	—	`milestone`<`CC`

ones); *(ii)* a set of *merging* rules, indicating when two or more contemporaneous events (with the same predominance level) must be merged together, and which activity label must be assigned to the resulting aggregated event; *(iii)* a set of *sort* rules, specifying an ordering relation over (non-purged and non-merged) simultaneous events.

Any combination of the above kinds of rules will be collectively regarded, hereinafter, as a *macro-event criterion*. The default instantiation of this criterion is summarized in Table 1, where each event is given a “predominance” level, only based on *what* attribute was updated in the event. Such levels acts as a sort of priority in the selection of events (the lower the level, the greater the priority): an event is eventually kept only if there is no simultaneous event with a lower level than it. In particular, events involving a change to the `status` or `resolution` hide `assignee`/`priority`/`severity` updates, which, in their turn, hide changes to the `milestone` or `CC`.

A *merging* rule is defined in Table 1 only for 1-level simultaneous events, which states that, whenever the `status` and `resolution` of a bug are modified contemporarily, the respective events must be merged into a single macro-event, labelled with the concatenation of their associated activity labels. For example, this implies that the ninth and tenth events in Figure 1 will be merged together, and labelled with the string “`state`:=*resolved* + `resolution`:=*fixed*”.

Also the default sort rule (shown in the table as an ordering relation $<$) only depends on the *what* field, and states that events involving attribute `milestone` must precede those concerning `CC`, and that `priority` (resp., `severity`) updates must precede `severity` (resp., `assignee`) ones. In this way, e.g., the first two events in Figure 1 will be switched with one another.

Example 1. Let us apply all default log abstraction operators introduced above (i.e. the event abstraction function in Equation 1 and the macro-event criterion of Table 1) to the bug trace τ_{ex} encoding the events in Figure 1. For the sake of conciseness, let us only consider events involving the attributes in Table 1. The resulting trace τ'_{ex} consists of 8 events, which are associated with the following activity labels, respectively: l_1=“Δ`milestone`”, l_2=“Δ`CC`”, l_3= “Δ`assignee`”, l_4= “Δ`CC`”, l_5= “Δ`CC`”, l_6= “Δ`assignee`”, l_7= “`status`:=*resolved*+`resolution`:=*fixed*”, l_8= “`status` := *closed*”, where l_i is the activity label of $\tau'_{ex}[i]$. The respective timestamps (at 1-hour granularity) of these events are: $t_1 = t_2 =$(*2012-06-20 10EDT*), $t_3 =$(*2012-06-20 22EDT*), $t_4 =$(*2012-06-21 11EDT*), $t_5 =$(*2012-06-25 05EDT*), $t_6 =$(*2012-07-02 10EDT*), $t_7 =$(*2012-07-02 15EDT*), $t_8 =$(*2013-01-03 11EST*). $\lhd$

State-oriented Trace Abstraction: For each trace τ, a collection of relevant prefixes (i.e. sub-traces) $rp(\tau)$ is selected, in order to extract an abstract representation for the states traversed by the associated bug, during its life. Two strategies can be adopted to this end, named *event-oriented* and *block-oriented*. In the former strategy all possible τ's prefixes are considered, i.e. $rp(\tau) = \{\tau(i] \mid i = 1 \ldots len(\tau)\}$, whereas in the latter only prefixes ending with the last event of a “macro-activity” are selected, i.e. $rp(\tau) = \{\ \tau(i] \mid 1 \leq i \leq len(\tau)$ and $when(\tau[j]) > when(\tau[i])\ \forall j \in \{i+1, \ldots, len(\tau)\}\ \}$.

Independently of the selection strategy, each trace τ' in $rp(\tau)$ is turned into a tuple $state^{\alpha}(\tau')$, whose attributes are all the abstract activities produced by a given event abstraction function α (e.g., that in Eq. 1). The value taken by each of these activities, say a, is denoted by $state(\tau')^{\alpha}[a]$ and computed as follows:

$$state^{\alpha}(\tau')[a] = \texttt{SUM}(\{\delta(\tau'[i]) \mid \alpha(\tau'[i]) = a, i = 1..len(\tau')\}) \quad (2)$$

where δ is a function assigning an integer weight to each event, based on its properties; by default it is *(i)* $\delta(e) = |added(e)|$, if e is not an aggregation of multiple simultaneous events (i.e. it corresponds to one raw update record) and e involves a multivalued attribute (like `CC`, `seeAlso`), or *(ii)* $\delta(e) = 1$ otherwise.

Any prefix trace τ' is hence encoded by an integer vector in the space of the abstract activities extracted by α, where each component accounts for all the occurrences, in τ', of the corresponding activity. Such a vector captures the state of a bug (at any step of its evolution) through a summarized view of its history.

Example 2. Let us consider the trace τ'_{ex} shown in Example 1. The unfolding of this trace gives rise to 8 distinct prefix sub-traces, denoted by $\tau'_{ex}(1]$, $\tau'_{ex}(2]$, …, $\tau'_{ex}(8]$. Five distinct abstract activities occur in these traces: a_1=“Δ`milestone`”, a_2=“Δ`CC`”, a_3=“Δ`assignee`”, a_4=“`status`:=*resolved*+ `resolution`:=*fixed*)”, a_5=“`status` :=*closed*”. As to trace

abstractions, all components of $state^{\overline{\alpha}}(\tau'_{ex}(1])$ (i.e. the tuple encoding the state reached after the first step) are 0 but that associated with a_1 and a_2, which are $state^{\overline{\alpha}}(\tau'_{ex}(1])[a_1] = 1$, and $state^{\overline{\alpha}}(\tau'_{ex}(1])[a_2] = 2$. — indeed, two values were added to CC in the first macro-activity. If using the event-oriented strategy, the above traces will generate 8 state tuples: $state^{\overline{\alpha}}(\tau'_{ex}(1])=\langle 1,2,0,0,0\rangle$, $state^{\overline{\alpha}}(\tau'_{ex}(2])=\langle 1,2,1,0,0\rangle$, $state^{\overline{\alpha}}(\tau'_{ex}(3])=\langle 1,4,1,0,0\rangle$, $state^{\overline{\alpha}}(\tau'_{ex}(4])=\langle 1,5,1,0,0\rangle$, $state^{\overline{\alpha}}(\tau'_{ex}(5])=\langle 1,5,2,0,0\rangle$, $state^{\overline{\alpha}}(\tau'_{ex}(6])=\langle 1,5,2,1,0\rangle$, $state^{\overline{\alpha}}(\tau'_{ex}(7])=\langle 1,5,2,1,1\rangle$. ◁

Such a state-oriented representation of a log L will be eventually exploited to induce a fix-time predictor (i.e. a $FTPM$) for L, as explained in the next section.

5 DISCOVERY APPROACH

We can now illustrate our whole approach to the discovery of a Fix-time Prediction Model ($FTPM$), based on a given set of raw bug records. The approach is illustrated in Figure 2 as a meta-algorithm, named FTPM Discovery, which encodes the main logical steps of our (process-oriented) data-transformation methodology, as well as the eventual application of a predictive induction method to the transformed log.

The algorithm takes as input a bug repository, storing a collection of bug records (like those described at the beginning of Section 3), along with a number of parameters concerning the application of data-manipulation operators.

In order to apply the abstraction operators introduced in Section 4, bug data are first turned into a set of bug traces (i.e. a bug log).

Based on a given *filtering* criterion Φ, function filterEvents is used to possibly remove uninteresting events (e.g., outliers or noisy data), which may confuse the learner, and lead to poor predictions.

Function handleMacroEvents allows us to apply a given *macro-event* criterion Γ (such as that described in Table 1) to rearrange each group of simultaneous log events according to the associated *predominance*, *reordering* and/or *merging* rules.

The two following steps (Steps 4 and 5) are meant to possibly associate each bug trace τ with additional "derived" data, in order to complement the original contents of $data(\tau)$ with context information. In fact, the insertion of such additional information was already considered in previous bug analysis works (Hooimeijer and Weimer, 2007; Marks et al., 2011), and was proven effective in improving the accuracy of predictive models (Bhattacharya and Neamtiu, 2011). Basically, function deriveTraceAttributes is devoted to insert new

Input: A collection B of bug records (cf. Section 3), a filtering criterion Φ, a macro-event criterion Γ, an event abstraction function α, and a prefix selection strategy $S \in \{$BLOCK, EVENT$\}$

Output: An $FTPM$ (Fix-time Prediction Model) for B

Method: Perform the following steps:

1 Convert B into a log L of bug traces;
2 $L :=$ filterEvents(L, ϕ);
3 $L :=$ handleMacroEvents(L, Γ);
4 $L :=$ deriveTraceAttributes(L);
5 $L :=$ refineTraceAttributes(L);
6 **if** $S =$ BLOCK **then**
7 RS:=$\{\tau(i]) \mid \tau \in L, 1 \leq i \leq len(\tau)$, and $when(\tau[j]) > when(\tau[i]) \; \forall j \in \mathbb{N}$ s.t. $i < j \leq len(\tau)\}$;
8 **else**
9 RS:= $\{\tau(i]) \mid \tau \in L$, and $1 \leq i \leq len(\tau)\}$;
10 **end if**
11 $M :=$ mineFTPM(RS, α);
12 **return** M.

Figure 2: Meta-algorithm FTPM Discovery.

derived trace attributes, defined as some summarized statistics over bug field/trace collections. Conversely, function refineTraceAttributes allows to transform a number of (raw or derived) bugs/events attributes, by turning each of them into a more expressive attribute. Two kinds of capabilities are provided by our framework to this end: *(i) attribute enrichment*, which consists in extending the values of an attribute with correlated information (extracted from the same repository), and *(ii) attribute aggregation*, which consists in reducing the dimensionality of an attribute by partitioning its domain into classes. Further details on the current implementation of both functions are presented in the next section.

Steps 6-10 are simply meant to extract a set RS of relevant (sub-)traces out of $\mathcal{P}(L)$, based on the chosen selection strategy S. RS is then used by function mineFTPM as a training set, in order to eventually induce an $FTPM$. To this end, as explained in Section 4, each trace $\tau \in RS$ is converted into a tuple labelled with the fix-time measurement $\mu_F(\tau)$, and encoding both the representation of τ's state (w.r.t. the given event abstraction function α), and its associated (augmented) data tuple $data(\tau)$. More precisely, $data(\tau)$ and $state^{\alpha}(\tau)$ are used as descriptive/input attributes, while regarding the actual remaining-time value $\mu_F(\tau')$ as the target of prediction.

At this point, a wide range of learning methods (including those described in Section 2) can be used to induce a regression or classification model. As a matter of fact, different solutions for carrying out this task are available in the current implementation of our approach, as described in detail in the next section.

6 PROTOTYPE SYSTEM

A prototype system was developed to fully implement the approach presented above, and support the discovery of high-quality *FTPM*s. In particular, the system allows the analyst to flexibly and modularly apply all the data-processing operators presented in this work, in an interactive and iterative way, as well as to define and store (in a reusable form) new variants of them, according to a template-based paradigm. The architecture of the system is shown in Figure 3.

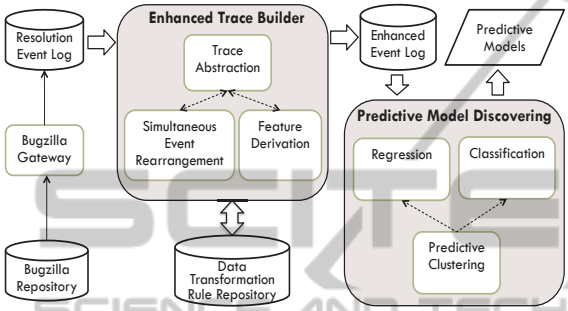

Figure 3: Logical architecture of the system prototype.

Module *Bugzilla Gateway* is capable to extract historical data stored in any *Bugzilla repository* (through its web interface), and to convert them into bug traces. The imported log, possibly cleaned (according to suitable filtering rules), is stored in the *Resolution Event Log*. Notice that, in principle, the system can be extended with analogous modules for importing log data from different bug tracking platforms.

The *Enhanced Trace Builder* module allows to apply various data-transformation criteria (possibly already stored in the *Data-Transformation Rule Repository*), in order to produce an enhanced version of the log, more suitable for fix time prediction. Specifically, the *Simultaneous Event Rearrangement* sub-module can be exploited to manipulate each group of simultaneous events according to some macro-event criterion, as discussed in Section 4. Conversely, the *Features Derivation* sub-module helps possibly enrich all bug traces with additional (derived and/or abstracted) context data. In any case, the *Trace Abstraction* block is eventually employed to build a state-oriented abstraction for each selected (sub-)trace.

Each "refined" log view obtained by way of the above described functionalities, and stored in the *Enhanced Event Log* repository, can be used as input for module *Predictive Model Discovering*. To this end, the abstracted traces are delivered to either the *Regression* or *Classification* module, based on which learning task was chosen by the user. Preliminary to the induction of a prediction model, the traces can be possibly partitioned into groups by one of the predictive clustering methods implemented by the *Predictive Clustering* module, which will also produce a set of decision rules for discriminating among the discovered clusters. In this case, each cluster will be eventually equipped with a distinct fix-time predictor.

Details on Built-in *Derived* and *Abstracted* Data: The following context data are automatically inserted by our system into each bug trace τ: *(i)* a collection of rough workload indicators, storing the overall number of bugs currently opened in the system, and the number of those pertaining the same product version (resp., component and OS) as τ; *(ii)* an analogous collection of counters for the bugs fixed in the past year (globally, and for the version/component/OS referred to by τ); *(iii)* a "reputation" coefficient, computed for the reporter of τ as in (Bhattacharya and Neamtiu, 2011); *(iv)* the average fix-time for various groups of related bugs (e.g., those concerning the same project or reporter as τ) and closed in the past year; *(v)* several seasonality dimensions (such as, week-day and month) derived from the date of the last event in τ.

As to the refinement of data, the system implements a specific attribute enrichment mechanism, which allows to replace users' identifiers (possibly appearing, e.g., in *who* fields of bug history records, or in certain bug attributes) with their respective e-mail addresses — actually, no further information on people is available in many real bug repositories. To this end, a greedy matching procedure was developed, based on comparing any user ID with all the email addresses appearing in various attributes of the bugs.

A further semi-automated built-in procedure available in the system allows instead to group the values of a given bug attribute *a*, by heuristically finding a partitioning that exhibits high correlation with fix-time values, based on a given aggregation hierarchy – such a hierarchy can be already available for the attribute (as in the case of email addresses and software products, which follow implicit meronomical and taxonomical schemes, respectively), or can be computed automatically (via an ad-hoc clustering approach). Essentially, the procedure tries to find an optimal cut of the hierarchy, looking at the information loss that is produced when real fix-time values are approximated with the averages computed over the selected nodes. Details are omitted for lack of space.

Details on Built-in Induction Methods: Several alternative learning methods are currently implemented in our system, which support the induction of a *FTPM*, from a propositional training set like that described in the previous section. These methods,

ranging from classical regression methods to state-aware Process Mining methods (van der Aalst et al., 2011; Folino et al., 2012; Folino et al., 2013), are listed next:

- `IBK`, a lazy (case-based) naïve regression method, implementing the k-NN procedure available in Weka (Tan et al., 2005), using $k = 1$ and Euclidean distance (and nominal attributes's binarization);
- `RepTree`, implementing the homonymous regression-tree learning method (Tan et al., 2005), while using the variance reduction criterion and 4-fold reduced-error pruning (as well as with a minimum value of 0.001 and 2 for the node variance and node coverage, respectively);
- `AFSM`, implementing the FSM-based learning method in (van der Aalst et al., 2011), using no history horizon and the multi-set trace abstraction (which yields the same state codes as Eq. 2 with unitary event weights, i.e. $\delta(e) = 1 \; \forall e \in E$);
- `CATP`, implementing the approach in (Folino et al., 2012), which first builds a multi-target predictive clustering for the bugs, using a greedy selection of all the partial fix-time values of each bug, and then equip each cluster with an *AFSM* prediction model (by reusing the previous method);
- `AATP-IBK` and `AATP-RepTree`, which combine a multi-target predictive clustering procedure with the base learners `IBK` and `RepTree`, respectively, following the approach in (Folino et al., 2013);
- `CBTP_1Reg` (standing for "Clustering Based Time Predictor with 1-dimensional Regression"), a novel method which first computes a regression tree by way of algorithm `RepTree`, using each bug as a single training instance, with its overall fix-time as target; a classic linear-regression model is then learnt for each cluster.

Like in previous bug analysis works, the analyst can also induce a classification model for the prediction of (discrete) fix times, after defining a set of a-priori classes in terms of fix-time ranges (possibly with the help of automated binning tools). To this end, a number of existing classifier-induction algorithms can be exploited, including the following ones:

- `J48`, the Weka's implementation of classical *C*4.5 (Quinlan, 1993) algorithm (with 3-fold reduced error pruning);
- `Random Forest`, implementing the algorithm in (Breiman, 2001) for inducing a random forest of decision trees (of size 10);
- `MRNB`, a two-phase induction method proposed in (Costa et al., 2009), which follows a sort of predictive-clustering strategy, where an initial rule-based classification model is refined by equipping each leaf with a probabilistic classifier.

7 CASE STUDY

This section discusses some tests performed with our prototype system, concerning the induction of different fix-time predictors from real data, extracted from the Bugzilla repository of project *Eclipse*. Two induction tasks were considered in the tests: *(i)* discover a *regression* model for predicting numeric fix-time values, and *(ii)* discover a *classification* model, w.r.t. a given set of time span classes.

Original Data and Derived Logs: A sample of 3906 bug records (gathered from January 2012 to March 2013) was turned into a set of bug trace like those described in Section 3. An explorative analysis of this log showed that the length of full bug traces ranges from 2 to 27, while bug fix time ranges from one day (i.e., a bug is opened and closed in the same day) to 420 days, with an average of about 59 days.

In order to make this log more suitable for prediction, we applied the basic event abstraction function $\overline{\alpha}$ of Eq. 1, so obtaining a first "refined" view L_0 over the selected bug traces.

Four further log views, named $L_1, \ldots, L_4$, were then derived from L_0, by incrementally applying the data-processing functions appearing in algorithm `FTPM Discovery` (cf. Figure 2).

A first cleaned view L_1, consisting of 2283 traces, was produced by applying to L_0 a specific instantiation of function `filterEvents`, removing the following data: *(i)* bugs never fixed, *(ii)* "trivial" bug cases (i.e. all bugs opened and closed in the same day), and *(iii)* trace attributes (e.g., `version`, `whiteboard`, and `milestone`) featuring many missing values, and bug/event fields (e.g. `summary`) containing long texts.

In order to take advantage of the restructuring of simultaneous events, a view L_2 was produced by treating L_1 with the default implementation of function `hanldeMacroEvents` (based on the rules of Table 1).

View L_3 was obtained applying a number of attribute derivation mechanisms available in our system (as a built-in implementation of function `deriveTraceAttributes`) to L_2.

L_4 was derived from L_3 through the built-in implementation of function `abstractTraceAttributes`. In particular, all people identifiers in the `reporter` bug attribute were replaced with a number of reporters' groups representing different organizational units (namely, {*oracle, ibm.us, ibm.no_us, vmware,*

Table 2: Regression results on Eclipse bug data. Rows correspond to different $FTPM$ induction methods, tested in two learning settings: without and with bug history events. Columns $L_0, \ldots, L_4$ correspond to different views of the original dataset, each obtained by a specific combination of pre-processing operations, as explained in the text.

Predictors		**rmse**					**mae**				
Setting	*Methods*	L_0	L_1	L_2	L_3	L_4	L_0	L_1	L_2	L_3	L_4
No Bug History (Baseline)	IBK	1.051	1.051	1.050	1.092	1.093	0.569	0.569	0.561	0.583	0.584
	RepTree	0.973	0.973	0.970	0.966	0.925	0.562	0.562	0.552	0.547	0.546
	Avg (no history)	**0.973**	**0.973**	**0.970**	**0.966**	**0.925**	**0.562**	**0.562**	**0.552**	**0.547**	**0.546**
History-aware	AFSM	1.123	1.027	1.010	1.010	1.010	0.717	0.640	0.647	0.647	0.647
	CATP	0.967	0.873	0.880	0.737	0.640	0.510	0.467	0.440	0.380	0.320
	IBK	0.983	0.823	0.807	0.793	0.803	0.430	0.360	0.360	0.347	0.360
	AATP-IBK	1.003	1.007	0.827	0.800	0.710	0.437	0.473	0.367	0.353	0.310
	RepTree	1.013	0.883	0.907	0.910	0.773	0.533	0.473	0.473	0.477	0.367
	AATP-RepTree	0.970	0.930	0.887	0.783	0.657	0.510	0.530	0.437	0.390	0.313
	CBPT_1Reg	0.947	0.900	0.750	0.700	0.547	0.490	0.453	0.383	0.350	0.280
	Avg (history-aware)	**1.001**	**0.920**	**0.867**	**0.819**	**0.734**	**0.518**	**0.485**	**0.444**	**0.420**	**0.371**

other}), and extracted semi-automatically from e-mail addresses. A similar approach was used to produce a binary abstraction (namely {*eclipse*, *not_eclipse*}) of attribute `assignee`, and an aggregate representation of both `product` and `component`.

Regression Results: When facing the prediction of fix times by way of regression techniques, prediction accuracy was evaluated through the standard error metrics *root mean squared error (rmse)* and *mean absolute error (mae)*. Both metrics were computed via 10-fold cross validation, and normalized by the average fix time (59 days), for ease of interpretation.

Table 2 reports the normalized *rmse* and *mae* errors obtained, with different numeric prediction methods available in our system (and described in Section 6), on the five log views described above. Two different learning setting were considered to this end: using bug history information (originally registered in terms of attribute-update events), and neglecting it. Note that the latter setting is intended to provide the reader with a sort of baseline, mimicking the approach followed by previous fix-time prediction works.

In general, it is easy to see that results obtained in the first setting ("no bug history") — where only the initial data of reported bugs are used as input variables for the prediction — are rather poor, if compared to the average ones obtained in the "history-aware" setting. Indeed, the errors measured in the former setting are quite high, no matter which inductive methods (i.e. `IBK` or `RepTree`) is used, and which combination of pre-processing operations are applied to the original logs. Interestingly, this result substantiates our claim that the exploitation of bug activity information helps improve the precision of fix-time forecasts.

On the other hand, in the second setting, both *rmse* and *mae* errors tend to decrease when using more refined log views. In particular, substantial reductions were obtained with the progressive introduction of macro-event manipulations (view L_2), and of derived and abstracted data (views L_3 and L_4, respectively).

By a finer grain analysis, we can notice that this trend is not followed by `AFSM`, which exhibits worse performances than the other history-aware methods, over all the log views. This bad behavior may be ascribed to the fact that `AFSM` does not exploit context data, which instead seem to be a key factor of improvement for fix-time prediction accuracy.

Very good results are obtained (in the history-aware setting) when using some kind of predictive clustering method, be it single-target (`CBPT_1Reg` and `RepTree`) or multi-target (`CATP`, `AATP-RepTree` and `AATP-IBK`). However, trace-centered clustering approaches (namely, `CATP`, `AATP-RepTree`, `AATP-IBK` and `CBTP_1Reg`) achieve better results than `RepTree`, which considers all possible trace prefixes for the clustering. In fact, the benefit of using a clustering procedure is quite evident in the case of `IBK`, which generally gets worse achievements than any other approach, presumably due to its inability to fully exploit derived data. Indeed, still focusing on the history-aware setting, it can be noticed that the prediction accuracy of `IBK` slightly increases when it is embedded in the predictive clustering scheme of `AATP-IBK`.

Classification Results: Let us finally show some of the results obtained by facing the discovery of a fix-time predictor as a classification problem, as commonly done in current literature. Two learning settings are considered again, based on the possibility to use bug-history data when inducing a classification model. The case where such data are disregarded still plays here a sort of baseline, corresponding to the approach followed in several fix-time prediction works (Giger et al., 2010; Marks et al., 2011).

Target classes were identified by discretizing – via equal-depth binning – the fix times of all bugs considered in the tests. These classes roughly correspond to

Table 3: Accuracy results of different fix-time classifiers on a fully enhanced log (L_4), derived from Eclipse bugs.

Predictors		Accuracy Measures		
Approach	*Methods*	*P*	*R*	F_1
No Bug History	J48	0.560	0.562	0.559
	MRNB	0.582	0.583	0.582
	Random Forest	0.538	0.541	0.539
	Avg (no history)	**0.560**	**0.562**	**0.560**
History-aware	J48	0.736	0.728	0.726
	MRNB	0.818	0.822	0.819
	Random Forest	0.815	0.816	0.815
	Avg (history-aware)	**0.790**	**0.789**	**0.787**

the following ranges: $\mu_F \leq 1$ day, 1 day $< \mu_F \leq 10$ days, 10 days $< \mu_F \leq 2$ months, and $\mu_F > 2$ months.

Table 3 reports the accuracy results obtained, against the most refined log view (i.e. L_4)[1], by three different induction methods (namely, J48, RandomForest and MRNB) implemented in our system. Three standard metrics were computed (via 10-fold cross-validation) to evaluate models' accuracy: *precision* (*P*), *recall* (*R*) and the balanced F_1 score (a.k.a. F-measure), defined as $F_1 = 2 \cdot P \cdot R \,/\, (P+R)$.

These figures confirm that the exploitation of bug history allows for improving neatly the accuracy of discovered models, regardless of the learning method and of the evaluation measure. In particular, very good scores are achieved when using (on history-aware logs) the Random Forest and MRNB methods.

8 CONCLUSIONS

A methodological framework for the prediction of bug fix times and an associated prototype system have been proposed, which fully exploit bug attributes' change logs. Provided with a rich collection of flexible data-transformation methods, the analyst can obtain a high-quality view of such logs, prior to applying Process Mining techniques to discover a process-aware prediction model. Encouraging results were obtained on some bug logs of a real open-source project, which empirically prove the benefits of exploiting bug update histories, and of employing our data manipulation methods.

As to future work, we plan to extend our approach in order to deal with long textual descriptions associated with bug/issue reports, as well as to predict other process-oriented performance measures than the sole fix time (e.g., QoS or cost indicators). We will also explore the application of our methods to the logs of other kinds of data-centric and lowly-structured collaboration environments (such as, e.g., issue-tracking and data-centered transactional systems).

[1]Less accurate models were extracted from the other (less refined) log views (namely, $L_0, \ldots, L_3$). Detailed results found in these cases are omitted for lack of space.

REFERENCES

Anbalagan, P. and Vouk, M. (2009). On predicting the time taken to correct bug reports in open source projects. In *Proc. of Int. Conf. on Software Maintenance (ICSM'09)*, pages 523–526.

Bhattacharya, P. and Neamtiu, I. (2011). Bug-fix time prediction models: can we do better? In *Proc. of 8th Intl. Conf. on Mining Software Repositories (MSR'11)*, pages 207–210.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Costa, G., Guarascio, M., Manco, G., Ortale, R., and Ritacco, E. (2009). Rule learning with probabilistic smoothing. In *Proc. of 11th Int. Conf. on Data Wareh. and Knowl. Discovery (DaWaK'09)*, pages 428–440.

Folino, F., Guarascio, M., and Pontieri, L. (2012). Discovering context-aware models for predicting business process performances. In *Proc. of 20th Intl. Conf. on Cooperative Inf. Systems (CoopIS'12)*, pages 287–304.

Folino, F., Guarascio, M., and Pontieri, L. (2013). A data-adaptive trace abstraction approach to the prediction of business process performances. In *Proc. of 15th Intl. Conf. on Enterprise Information Systems (ICEIS'13)*, pages 56–65.

Giger, E., Pinzger, M., and Gall, H. (2010). Predicting the fix time of bugs. In *Proc. of 2nd Intl. Workshop on Recommendation Systems for Software Engineering (RSSE'10)*, pages 52–56.

Hooimeijer, P. and Weimer, W. (2007). Modeling bug report quality. In *Proc. of 22nd IEEE/ACM Intl. Conf. on Automated Software Engin. (ASE'07)*, pages 34–43.

Marks, L., Zou, Y., and Hassan, A. E. (2011). Studying the fix-time for bugs in large open source projects. In *Proc. of 7th Intl. Conf. on Predictive Models in Software Engineering (Promise'11)*, pages 11:1–11:8.

Panjer, L. (2007). Predicting eclipse bug lifetimes. In *Proc. of 4th Intl. Workshop on Mining Software Repositories (MSR'07)*, pages 29–.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Tan, P.-N., Steinbach, M., and Kumar, V. (2005). *Introduction to Data Mining*. Addison-Wesley Longman.

van der Aalst, W., van Dongen, B., Herbst, J., Maruster, L., Schimm, G., and Weijters, A. (2003). Workflow mining: a survey of issues and approaches. *Data & Knowledge Engineering*, 47(2):237–267.

van der Aalst, W. M. P., Schonenberg, M. H., and Song, M. (2011). Time prediction based on process mining. *Information Systems*, 36(2):450–475.