Fixing Bugs in Your Sleep: How Genetic Improvement Became an Overnight Success

Saemundur O. Haraldsson*
University of Stirling
Stirling, United Kingdom FK9 4LA
soh@cs.stir.ac.uk

John R. Woodward
University of Stirling
Stirling, United Kingdom FK9 4LA
jrw@cs.stir.ac.uk

Alexander E.I. Brownlee
University of Stirling
Stirling, United Kingdom FK9 4LA
sbr@cs.stir.ac.uk

Kristin Siggeirsdottir
Janus Rehabilitation Centre
Reykjavik, Iceland
kristin@janus.is

ABSTRACT

We present a bespoke live system in commercial use with self-improving capability. During daytime business hours it provides an overview and control for many specialists to simultaneously schedule and observe the rehabilitation process for multiple clients. However in the evening, after the last user logs out, it starts a self-analysis based on the day's recorded interactions. It generates test data from the recorded interactions for Genetic Improvement to fix any recorded bugs that have raised exceptions. The system has already been under test for over 6 months and has in that time identified, located, and fixed 22 bugs. No other bugs have been identified by other methods during that time. It demonstrates the effectiveness of simple test data generation and the ability of GI for improving live code.

CCS CONCEPTS

•Software and its engineering → Error handling and recovery; Automatic programming; Maintaining software; Search-based software engineering; *Empirical software validation;*

KEYWORDS

Genetic Improvement, Adaptive System, Bug fixing, Test data generation

ACM Reference format:
Saemundur O. Haraldsson, John R. Woodward, Alexander E.I. Brownlee, and Kristin Siggeirsdottir. 2017. Fixing Bugs in Your Sleep: How Genetic Improvement Became an Overnight Success. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082517

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO '17 Companion, Berlin, Germany
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-4939-0/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3067695.3082517

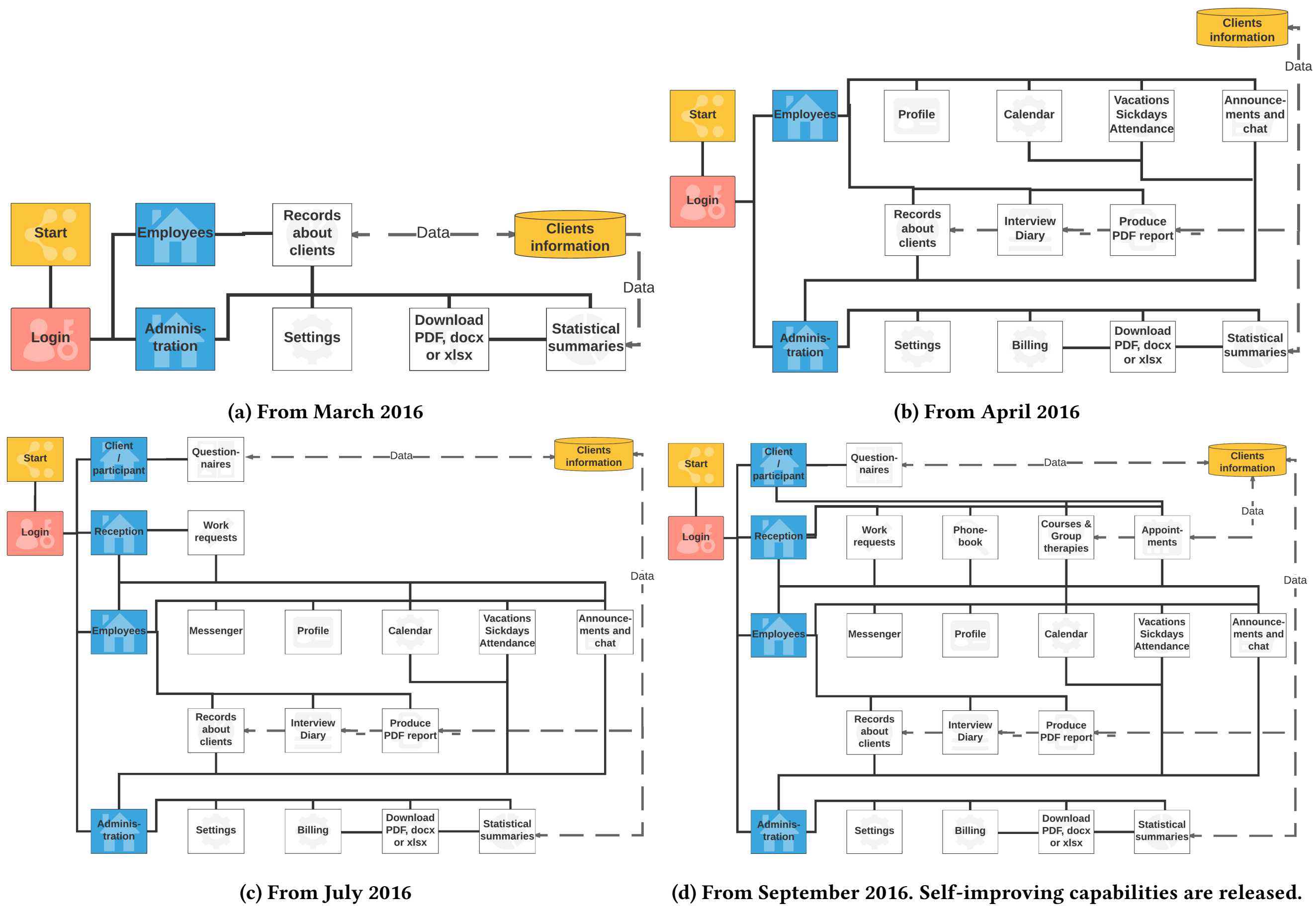
1 INTRODUCTION

Genetic Improvement (GI) [38] is a growing area within Search Based Software Engineering (SBSE) [23, 24] which uses computational search methods to improve existing software. Despite its growth within academic research the practical usage of GI has not yet followed. Like with many SBSE applications, the software industry needs an incubation period for new ideas where they come to trust in outcomes and see those ideas as cost effective solutions. GI is in the ideal position to shorten that period for the latter as it presents a considerable cost decrease for the software life cycle's often most expensive part: maintenance [18, 34]. There are examples of software improved by GI being used and publicly available [31] which is impressive considering how young GI is as a field. In time it can be anticipated that we will see tools emerging that utilise current advances in GI for various improvements during different stages of development; from early coding where programmers might want to statically monitor the performance of their work to maintenance on bug fixing [2], automatically adding new functionality [36], and adding new hardware compatibility [28].

Traditionally, GI applications have been used offline where the program is copied, improved in a lab, and then the researchers have to convince developers to include the improvements in later releases, if the researchers have the patience. However GI's ultimate goal must be self-improving and self-adaptive systems [8, 10, 20] with minimal manual effort by the developers to truly optimise software maintenance costs. Current research into self-improving systems consists of early concepts [7, 46], identifying applicable ideas [22], and highly specified but truly dynamic approach [51]. The research and methods are slowly approaching dynamic adaptive systems but a big obstacle is getting the results to market. To achieve that we need to take it in steps, first we provide developers with tools before we provide autonomous software.

In this paper we describe a live system, Janus Manager (JM), that takes that first step. It monitors itself during daytime use and collects data whenever user input raises exceptions. Overnight it uses the collected data to test itself and searches for adaptions that do not raise exceptions when similar data is submitted later. At the end of the nightly self-improvement process it presents the developers with options to fix the perceived fault.

Our approach significantly decreases the cost of maintenance after the initial release but allows developers to ultimately have the control and provide a sanity check before patches are issued to the



(a) From March 2016

(b) From April 2016

(c) From July 2016

(d) From September 2016. Self-improving capabilities are released.

Figure 1: JM's feature map as it developed during the software's first 8 months in use. It shows how rapidly features were added after employees of JR started using the software in March 2016. This caused buggy and less well-tested code to be released, hence inspiring the integration of GI.

live software. JM is a bespoke program for a vocational rehabilitation centre, developed and maintained by Janus Rehabilitation (JR) in Reykjavik, Iceland [43, 44] and has been briefly described in previously published work [18, 19].

The remainder of the paper is structured as follows. Section 2 lists some related work and inspirations. Section 3 details what the system does during business hours and how it keeps records for later improvements. Section 4 explains how the daily data is used to improve the software system including generating test cases and the GI stage. Section 5 summarises the current data gathered since the launch of JM. Section 6 gives an overview of what future directions we are currently contemplating.

2 RELATED WORK

GI has most commonly been used offline to improve various software properties [15, 38]. More than a third of GI work has showcased non-functional improvements [47] such as Langdon et al.'s work on execution time with Bowtie2 [27], CUDA code [28, 29, 33], and Wu et al.'s deep parameter optimisation for memory consumption [50]. Other examples of execution time optimisations include other bioinformatics programs [17], software like satisfiability solvers [39, 40], various Unix programs [48, 50], and sort functions [9]. Progress in mobile technology and the need to preserve resources has also motivated work on energy usage optimisation [5, 6, 16, 49]. Although non-functional properties have been a prominent target for GI to tackle, bug fixing is by far the largest single problem in the GI literature [47] to be addressed. The work on bug fixing has led to the development of the well known tool, GenProg [35] and the discrete fitness function of passed test cases has motivated more fundamental work on GI search landscapes [19, 32]. Functionality improvements also include growing and grafting [21, 29, 36], repairing and optimising the distribution of hashcode implementations [25] and prediction model improvements [14].

As GI is a fairly young field the work on self-improving software with GI is not extensive, although SBSE literature has been considering self-adaptive systems for some time [8, 10, 20]. A few examples of adaptive or dynamic GI include a framework (Gen-O-Fix [46]) to continuously improve software on Java Virtual Machines in parallel with usage, Burles at al.'s list of suggestions for embedded

improvement methods [7], and an approach (ECSELR [51]) that injects dynamic adaptability in an already running target software. However, JM's approach is inspired by Harman et al.'s suggestion for *Dreaming Devices* [22], exploiting the fact that the majority of software is not in continuous use. Even if that were the case, the usage load will usually be periodic and during lower load times the device should be able to afford some capacity for improvements.

The common theme is that GI can and should be used to make dynamic adaptive software but the differences are the implementations and applications. JM marries the concept of the *Dreaming Device* with Arcuri's co-evolutionary bug fixing [2, 3] and further adds usage evolution.

In addition to GI we have implemented a Search Based Software Test (SBST [37]) method of test data generation, a simple random sampling of test data. Ideally, test data generation is about maximum code coverage and minimum redundancy. It is simply not worth adding test cases to test suites if they do not add to the coverage. The SBST literature has many solid examples of test data generation like uniform sampling with Boltzmann samplers [11], generating strings [4] for the maximisation of code coverage, or more specialised searches for data with specific properties [12, 41]. The search methods include alternatives such as hill-climbing [45], an Evolutionary Algorithm [26] or other optimisation algorithms [13].

Generated test data in JM is an emulation of actual inputs from a graphical user interface (GUI) [19]. There are however examples of work that generate test data for GUI testing by representing input fields with symbolic alternatives [42]. That, however, demands that the developer knows in better detail about how the software will be used, which in our case is near impossible since every client's route through the rehabilitation is probably unique.

3 JANUS MANAGER DAILY ACTIVITY

JM's creation was initially motivated by JR's need for specialised data management and statistical analysis for a rehabilitation service. Its functionality was simple to begin with (Figure 1a) but the lack of specialised software for complete management of rehabilitation services further drove the development (Figures 1b – 1c) to produce today's version as seen in Figure 1d. It is a software system that supports the vocational rehabilitation process and internal communications.

Moreover, JM is a tool for the directors to be able to continuously improve the rehabilitation process with statistical analysis of client data and performance of methods and approaches. It has to manage multiple connections between users, specialists and clients.

3.1 Usage of Janus Manager

The left side of Figure 2 displays the daytime normal usage of JM. The users are all employees of JR, over 40 in total, including both specialists and administrators. They interact with JM by either requesting or providing data which is then processed and saved. Example request are: Internal communications between the interdisciplinary team of specialists about clients, a journal record from a meeting, or an update to some information regarding the client. The system can also produce reports and bills in PDF format or rich text files (see Figure 1).

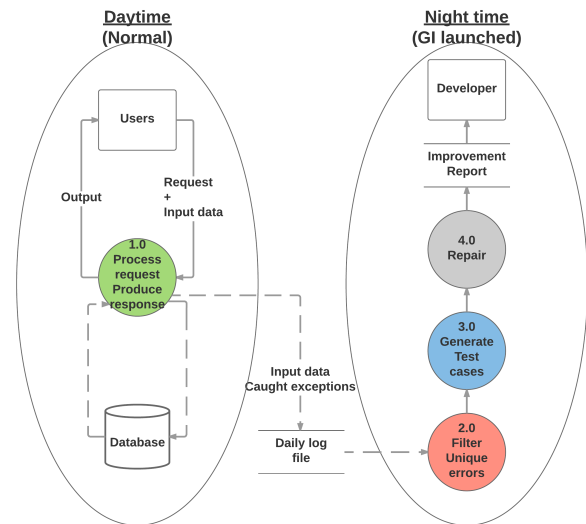


Figure 2: JM functionality divided into daytime processes and night-time processes.

The clients have access to specialised and standardised questionnaires that measure various aspects of the clients welfare and progress. The specialists then use the results from those questionnaires to plan a treatment or therapy. Additionally it uses the complete database of information to identify risk factors, and predict treatment outcome and length [14].

Every time an input data causes an unintentional exception to be thrown, JM logs the trace, input data and the type of exception in a daily log file shown in the middle of Figure 2.

3.2 Structure of Janus Manager

JR provides individualised vocational rehabilitation and as such users of JM regularly encounter unique use cases. Therefore JM is in active development while being in use. Features are continually added, based on user experience, feedback and convenience. The average growth of the system, in the first months following its initial release, was approximately 20 lines of code (LOC) per week. Six months later, in the weeks preceding the integration of the GI, the growth had decreased to 15 LOC per week and has been relatively stable since. Currently the system is over 25K lines of Python 2.7 (300 classes and 600 functions). Functions are on average 26 LOC and classes 36, ranging from 2 to 251 and 918 respectively. Each function and class has their own test suite, with a few exceptions[1]. The test suites are also of various sizes, depending on the functionality of the entity under testing.

JM runs as a web service on an Apache server running on a 64 bit Ubuntu with 48GiB RAM and two 6 core Intel processors. The GUI is an HTML web page that JM serves up from pre-defined templates.

JM's structure made the integration of its nightly activity as simple as wrapping the whole system in a single *try* and *except* statement that catches all exceptions that are not accounted for.

[1]Some base classes are never used directly so they do not have any tests

The modularity gave the GI easy access to test and improve each component in isolation.

4 JANUS MANAGER NIGHTLY ACTIVITY

After the last user logs off in the evening the nightly routine initiates (Figure 2, right side). The process runs until all reported bugs are fixed or until the next morning. During the night JM analyses the logs, generates new test data and uses GI to fix bugs that have been encountered during the day. The identified bugs are not necessarily faults in the program itself, rather a result of the developer's inability to account for all possible use cases of the system. On all occasions when the nightly activity was invoked and had bugs to fix it was because JR's employees had used JM in a way that had not been foreseen.

4.1 Log analysis

Going through the daily logs involves filtering the exceptions to obtain a set of unique errors in terms of exception type and location in the source code. The input is defined as the argument list at every function call on the trace route from the users' request to the location of the exception. The type of the exception can be any subclass of *Exception* in Python, both built in and locally defined.

The exceptions are sorted in decreasing order of importance, giving higher significance to errors that occurred more often, arbitrarily choosing between draws. This measure of importance assumes that these are use case scenarios that happen often and are experienced by multiple users and not a single user who repeatedly submits the same request.

4.2 Generating test data

The test data generation is done with a simple random search of the neighbourhood of the users' input data retrieved from the log entry. The input is represented by a Python *dictionary* object, where elements are (key, value) pairs and the values can be of any type or class. However, most types are strings, dates, times, integers or floating point numbers. The objective of the search is to find as many versions of the input data as possible that trigger the same exception. Procedure 1 details the search for new test data.

Starting with the original input θ we make 100 instances of $\theta^{mutated}$ where a single value has been randomly changed. For each instance the value to be mutated is randomly selected while all other values are kept fixed. Every $\theta^{mutated}$ that causes the same exception as the original is kept in Θ, others are discarded. Different exceptions are not considered since the setup looks for the specific exception from the log rather than any general exception. This is then repeated by randomly sampling from the latest batch of $\theta^{mutated}$, Θ^{latest} (see line 10) until either no new instances are kept or the maximum of 1000 instances have been evaluated (line 5).

The mutation mechanism in line 11 first chooses randomly between key, value pairs in θ^r only considering pairs where values are of type string, date, time, integer or float. Then depending on the type, the possible mutations are the following:

- **String** mutations randomly add strings from a predefined dictionary with white space and special characters, keeping the original as a sub-string

Procedure 1 Test data search

1: $\Theta \leftarrow [\theta]$ {Start with the original input}
2: $n \leftarrow 0$
3: $\Theta^{new} \leftarrow [\theta]$
4: $\Theta^{latest} \leftarrow []$
5: **while** ($n < 1000$) AND ($|\Theta^{new}|! = 0$) **do**
6: extend Θ with Θ^{latest}
7: $\Theta^{latest} \leftarrow \Theta^{new}$
8: $\Theta^{new} \leftarrow [\ \ ]$
9: **for** $i = 1$ until $i == 100$ **do**
10: $\theta^r \leftarrow$ random choice Θ^{latest}
11: $\theta^{mutated} \leftarrow$ mutate θ^r
12: **if** $\theta^{mutated} \rightarrow$ causes exception **then**
13: append $\theta^{mutated}$ to Θ^{new}
14: **end if**
15: $n+ = 1$
16: **end for**
17: **end while**

- **Date** mutations can change the format (e.g. 2017-01-27 becomes 27-01-17), the separator, or randomly pick a date within a year from the original
- **Time** mutations can change the format (e.g. 7:00 PM becomes 19:00), the separator or randomly pick a time within 24 hours from the original
- **Integer** mutations add or subtract 1, 2 or 3 from the original. Maximum of ±3 variation was arbitrarily chosen as a starting point for integer mutations because we assume integer inputs will not deviate much more from what is being observed from the user.
- **Float** mutations change the original with a random sample from the standard normal distribution $N(0, 1)$

All of the instances in Θ along with the original θ are then the inputs of the new unit tests. The assertion for each instance will check that the response is of the specific exception type and the tests will fail if the input triggers that exception. The new unit tests are then added to the existing test suite, automatically expanding the library of test cases.

There are mainly two problems with this approach; *a)* it does not check whether new test cases are complementary or not, i.e. if the two or more test cases are validating the same part of the code, and *b)* it assumes that if the exception is not raised then the output, if any is expected, is correct. The first problem is trivial when computing power is not an issue or if testing is not impeding development. The second problem is more serious because we cannot guarantee that the assumption holds and it might give false confidence to developers and wrong fitness evaluation to the GI. However the implementation of the whole system should at least catch any mistake the GI could introduce with these tests by passing the responsibility for sanity checks to the developers.

4.3 Genetic Improvement

The GI part of the overnight process relies on the new test cases in conjunction with a previously available test suite. The assumption is that, given the test suites, the program is functioning correctly if

Table 1: Sets of single operators available to the GI. One member of a given set can be changed to another member of the same set.

Description	Operations
Numerical constants	Can increment by ± 1
Arithmetic operators	$+, -, *, /, //, \%, **$
Arithmetic assignments	$+=, -=, *=, /=,$
Relational operators	$<, >, <=, >=, ==, !=,$ *is*, *is not*, *not*
Logical operators	*and*, *or*
Logical constants	*True*, *False*

it passes all test cases and so is awarded highest fitness. Otherwise fitness is proportional to the number of test cases the program passes of the whole suite.

The mutation process is inspired by Langdon et al.'s work [30] by evolving edit lists that operate on the source code (Figure 3). Each edit consists of: the operation of *Replace*; the source code snippet before and after the edit; and the location of where to apply this edit (line and character number). The edit lists define the operations replace, delete, and copy for code snippets, lines, and statements. The GI process operates on the source code with no need to convert the program to a different representation like abstract syntax trees (AST)[1]. Therefore it is directly transferable between programming languages with minimal configuration. The source code is read as a text file and stored in a data structure (x) of program lines.

Each line's data structure holds the following information:

- The raw text as it appears in the source file.
- Line type e.g. *function definition* or *if clause*.
- Indentation as the number of space characters[2].
- If the following lines should be indented or not.
- Whether the line can be altered or not. This can be varied by including or excluding certain line types in a list of untouchable line types. Empty lines, function and class definitions, imports, multiple line comments, and a few others are included by default.
- A list of variables, and operators from table 1 that the line contains, along with their location on the line. Regular expression patterns are used to identify and locate the operators. The patterns are kept simple to minimise constrictions to the search so the identification is vulnerable to false positives and might make changes available to the GI that do not always make sense. For an example the GI might find the operator < in a string constant and change it to <= which possibly has no effect on the fitness.

The evolution is population based with 50 edit lists (solutions) in each generation. Each generation is evaluated in parallel to minimise GI's execution time and to utilise the full power of the server. Weighted random selection is used to select parents for the next generation, i.e. each lists' weight determines how likely it is to be selected. The weight is determined by the edit list's proportional fitness with respect to the generation's total fitness. Only half of the population gets selected and they undergo mutation to start the next generation. Crossover is not used in the current implementation as well as elitism. The other half of the subsequent generation are randomly generated new edit lists. This selection method should deter homogeneity in the population and early convergence to a local optimum. However, it might also prevent the GI in finding solutions that require more than a few edits where any subset has poorer fitness than the original.

[2] Code blocks are defined by indentation in Python and not by {} as in Java/C

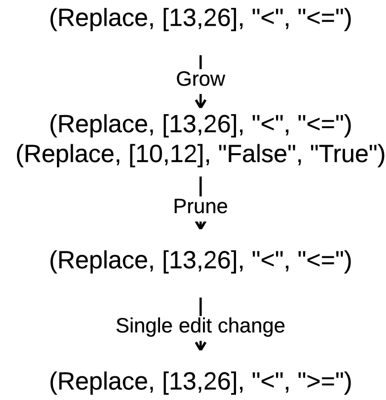


Figure 3: An example of an edit list and how it can evolve with *Grow*, *Prune* or *Single edit change*.

A single mutation of an edit list can be made with any of the examples in Figure 3. The options for mutations are:

Grow is where a randomly generated edit is appended to the edit list. The edit is generated by a stochastic selection from all possible locations in the source with a equal probability. The location can be a line and a column or just a line. If the case is the former, then another uniform random selection is made from possible replacements for the content of the location (see Table 1). For the latter case, either one or two more selections are made. First a selection of what operation will be applied to the selected line; *delete*, *replace*, *copy* or *swap*. The random edit build stops here if *delete* is selected. For replace and swap another line of same type is randomly selected with equal probability to be the replacement or the line that swaps places. For copy a random line number in the source is selected to be the location above which the selected line is copied to.

Prune is when an edit in the individual selected with uniform random distribution and every subsequent edit in the list is removed.

Single edit change is perhaps the least disruptive mutation. A single edit is selected and one of its features is randomly changed, such as the replacement code is re-selected or the copy location is altered.

Table 2: A list of the bugs that were automatically detected in JM and fixed by the GI. They are categorised by the type of exception that they caused and given an identification number in the second column.

Exception type	Id	Invocations	New test cases	Input type that caused exception	Suggested fix size (Accepted fix size)	Mean edit list size	Number of edit lists considered
IndexError	E1	6	1	Date tuple	2 (2)	2.53	412
	E2	3	1	Integer	3 (3)	2.48	356
	E3	3	2	Integer	2 (2)	2.38	367
	E4	1	1	Integer	4 (4)	2.62	437
TypeError	E5	36	1	Integer	3 (3)	2.23	426
	E6	6	3	String	4 (3)	2.41	465
	E7	1	1	Integer	2 (2)	2.67	457
	E8	2	1	(String, Integer)	1 (1)	2.50	442
	E9	2	1	String	5 (3)	2.50	413
	E10	1	1	String	2 (2)	2.47	412
UnicodeDecode Error	E11	4	1	String	2 (2)	2.48	424
	E12	3	2	String	3 (3)	2.48	404
	E13	2	2	String	2 (2)	2.48	465
ValueError	E14	4	2	Date and time	1 (1)	2.54	435
	E15	4	1	Date and time	1 (1)	2.57	388
	E16	3	1	String	3 (3)	2.44	428
	E17	3	1	Integer	5 (4)	2.49	353
	E18	2	1	Date and time	3 (3)	2.49	467
	E19	2	2	Date and time	3 (3)	2.40	405
	E20	1	1	Time	4 (3)	2.39	477
	E21	1	1	String	2 (2)	2.47	371
	E22	1	1	String	1 (1)	2.56	478

The GI only stops if it has found a program variant that passes all tests or just before the users are expected to arrive to work. It then produces an HTML report detailing the night's process for the developers. The report lists all exceptions encountered, new test cases and a list of possible fixes, recommending the fittest. If more than a single fix is found, then the report recommends the shortest in terms of number of edits. However it is always the developers choice to implement the changes as they are suggested, build on them or discard them.

5 SUMMARY

Development on JM started as a small in-house data management project by JR in March 2016. However, as JR's employees started using the software they identified multiple ways to enhance JM to improve productivity and efficiency in the rehabilitation process. JM's development has since been user-driven and quite rapid, with a new feature being added weekly in spring and summer 2016. While JM's size has increased the rate of new features has not decreased but they have become subjectively smaller, i.e. small from the user's perspective but not necessarily for the developer. As JR's main operation is vocational rehabilitation and not developing software, its core development team is minimal. This, combined with JM's rate of expansion caused poorly tested code to be repeatedly released and subsequently occupying the development team with bug fixing instead of further enhancing JM in a meaningful way. Therefore JM has been running its self-healing processes every night with exceptional results since September 2016. The integration of GI has roughly halved JR's maintenance cost for the system. Before the developers had to manually find and fix each bug but now they only have to validate the suggestions that are handed to them by the GI.

In the six months succeeding the launch of GI within JM, 22 bugs have been identified and fixed. During that time there have been no other bugs found or fixed by other methods after any version update, excluding bugs that were discovered by developers before updates where released. Table 2 lists the bugs that have been encountered, categorised by the exception type and order in descending order of how often each bug was invoked by users (Column 3). The table also lists how many new test cases, without duplicates, were added to the test suite (Column 4). In total 29 unique test cases have been added to JM's test suites but initial number of automatically generated test cases was 408. The developers could easily and swiftly discard duplicate test cases by hand.

The fifth column describes the input types that caused the exceptions. In most cases the types listed there are a single variable from an array of inputs but only the variables that were directly involved in throwing the exception are mentioned. Column six lists how many edits each suggested and accepted fix contained. In majority of cases the suggested fix was accepted as it was but on three occasions a single neutral edit was removed before accepting the fix (*E6*, *E17*, *E20*) and two edits from *E9*. All removed neutral edits were either duplicating a line or a variable, without it having effect on output. In four instances, either a single edit was slightly altered or a small manual edit to the source code was applied post-hoc (*E1*, *E15*, *E16*, *E22*). The last two columns contain the average size of

all edit lists that were evaluated for each bug and how many were evaluated, respectively.

If we look at the seventh column, we see that the average edit list size is nearly the same for all fixes ([2.23, 2.67]), which is to be expected since same search parameters were used in every case. On closer inspection, one way ANOVA reveals that the mean size of all evolutionary runs is most likely equal ($p > 0.9$). Furthermore, we see that the evolution never exceeds 10 generations (500 evaluated edit lists) and consequently the maximum limit of edit list size was 10 edits. Given that the average size of each function being fixed is 26 LOC the search space is relatively small so if a fix exists we expect to find it rather quickly.

A typical fix replaced a single line with a similar line from elsewhere in JM, like *E15* replaced:[3]

```
dum.occurance = \
   datetime.datetime.combine(dum.expected_occurance,\
   datetime.datetime.strptime(form['occurance'],\
   '%H:%M').time())
```

with:

```
dum.occurance = \
   datetime.datetime.combine(dum.expected_occurance,
   datetime.datetime.strptime(form['occurance'],\
   '%H:%M:%S').time())
```

The only difference is that the latter expects seconds to be included in the time format. The human programmer recognises it as a single edit of adding `:%S` but the GI replaced the whole line.

Another example is *E20*, in a function that checks for reoccurringly available meeting spaces on given weekdays. The bug was that the user sometimes omitted the time of day to be checked. The accepted fix wrapped the line obtaining the time argument in a *try* clause so this:

```
the_time = datetime.datetime.strptime(\
request.args.get('the_time','08:00'),\
'%H:%M').time()
```

became this:

```
try:
 the_time = datetime.datetime.strptime(\
 request.args.get('the_time','08:00'),\
 '%H:%M').time()
except ValueError:
 the_time = datetime.time(8,0)
```

Three edits of "copy line x above line y" were needed to accomplish this fix, the edit that was removed, duplicated the last line.

6 FUTURE WORK

JM, as introduced in this paper is fully implemented and live, although it has been so for a few months, it still needs to be tested further. Finding and fixing 22 bugs is impressive but we would like to do more. Our current task list includes but is not limited to:

- Integrate the GI in another software system. The GI is a standalone feature that is easily integrated in most software systems so a natural next step would be to identify services and systems where it could be of use.
- Improve the developer's interface with the GI. Current implementation reports only fixes that pass all tests but we might want to consider lesser variations. By presenting the developer with some good but not perfect solutions might provide additional information that they can use to fix the bug more effectively
- Improve the search ability. Truncated selection might inhibit larger edit lists to be evolved and possible multi-edit solutions are therefore lost. Parameter tuning is our initial step forward in this task.
- Improve the test data generation mechanism. Ideally we want to predict expected inputs to the system and possibly generate test data that imitates unseen future inputs. Also, we would like to improve the search process for new test data by implementing something other than random search. That includes, changing the fitness function (currently binary), adding more objectives and improving the sampling methods.

However a continuous task will be to monitor the JM system while it is being developed further and gather data on the bugs that are caught and fixed.

[3] The character \ denotes line continuation and is only used here for aesthetic purposes

ACKNOWLEDGMENT

The work presented in this paper is part of the DAASE project which is funded by the EPSRC Grant EP/J017515/1. The authors would like to thank JR for the collaboration and providing the platform which made the development possible.

REFERENCES

[1] T. Ackling, B. Alexander, and I. Grunert. Evolving Patches for Software Repair. In *GECCO'11, 13th annual conference on Genetic and evolutionary computation*, pages 1427–1434, Dublin, Ireland, jul 2011. ACM.

[2] A. Arcuri. On the Automation of Fixing Software Bugs. In *ICSE Companion '08 Companion of the 30th international conference on Software engineering*, pages 1003–1006, Leipzig, Germany, 2008. ACM.

[3] A. Arcuri, D. R. White, J. Clark, and X. Yao. Multi-Objective Improvement of Software using Co-evolution and Smart Seeding. In *Proceedings of the 7th International Conference on Simulated Evolution and Learning (SEAL'08)*, pages 1–10, 2008.

[4] M. Beyene and J. H. Andrews. Generating String Test Data for Code Coverage. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 270–279. IEEE, 2012.

[5] B. R. Bruce. Energy Optimisation via Genetic Improvement A SBSE technique for a new era in Software Development. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 819–820, Madrid, Spain, jul 2015. ACM.

[6] B. R. Bruce, J. Petke, and M. Harman. Reducing Energy Consumption Using Genetic Improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1327–1334, Madrid, Spain, jul 2015. ACM.

[7] N. Burles, J. Swan, A. E. Brownlee, E. Bowles, Z. A. Kocsis, and N. Veerapen. Embedded Dynamic Improvement. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion -15, pages 831–832, Madrid, Spain, jul 2015. ACM.

[8] B. H. C. Cheng, et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for SelfAdaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[9] B. Cody-kenny, E. Galván-lópez, and S. Barrett. locoGP : Improving Performance by Genetic Programming Java Source Code. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 811–818, Madrid, Spain, jul 2015. ACM.

[10] R. de Lemos, et al. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, volume 7475 of *Lecture Notes in Computer Science*, pages 1–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[11] P. Duchon and G. Louchard. Boltzmann Samplers For The Random Generation Of Combinatorial Structures. *Combinatorics Probability and Computing*, 13(4-5):577–625, 2004.

[12] R. Feldt and S. Poulding. Finding Test Data with Specific Properties via Metaheuristic Search. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 350–359. IEEE, 2013.

[13] R. Feldt and S. Poulding. Broadening the Search in Search-Based Software Testing: It Need Not Be Evolutionary. *Proceedings - 8th International Workshop on Search-Based Software Testing, SBST 2015*, pages 1–7, 2015.

[14] S. O. Haraldsson, R. D. Brynjolfsdottir, J. R. Woodward, K. Siggeirsdottir, and V. Gudnason. The Use of Predictive Models in a Dynamic Planning of Treatment. In *Proceedings - IEEE Symposium on Computers and Communications*, Heraklion, Greece, 2017. IEEE.

[15] S. O. Haraldsson and J. R. Woodward. Automated Design of Algorithms and Genetic Improvement : Contrast and Commonalities. In *Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion*, GECCO Comp '14, pages 1373–1380, Vancouver, Canada, jul 2014. ACM.

[16] S. O. Haraldsson and J. R. Woodward. Genetic Improvement of Energy Usage is only as Reliable as the Measurements are Accurate. In *Proceedings of the 2015 Conference Companion on Genetic and Evolutionary Computation Companion*, pages 831–832, Madrid, 2015. ACM.

[17] S. O. Haraldsson, J. R. Woodward, A. E. Brownlee, A. V. Smith, and V. Gudnason. Genetic Improvement of Runtime and its fitness landscape in a Bioinformatics Application. In *Proceedings of the 2017 Conference Companion on Genetic and Evolutionary Computation Companion*, Berlin, Germany, 2017. ACM.

[18] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, and D. Cairns. Exploring Fitness and Edit Distance of Mutated Python Programs. In *Proceedings of the 17th European Conference on Genetic Programming, EuroGP*, Amsterdam, The Netherlands, 2017. Springer Berlin Heidelberg.

[19] S. O. Haraldsson, J. R. Woodward, and A. I. E. Brownlee. The Use of Automatic Test Data Generation for Genetic Improvement in a Live System. In *8th International Workshop on Search-Based Software Testing*, Buones Aires, 2017. ACM.

[20] M. Harman, E. Burke, J. A. Clark, and X. Yao. Dynamic adaptive search based software engineering. In *International Symposium on Empirical Software Engineering and Measurement*, pages 1–8, Lund, Sweden, 2012. ACM.

[21] M. Harman, Y. Jia, and W. B. Langdon. Babel Pidgin : SBSE Can Grow and Graft Entirely New Functionality into a Real World System. In *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 247–252, Fortaleza, Brazil, aug 2014. Springer International Publishing.

[22] M. Harman, Y. Jia, W. B. Langdon, J. Petke, I. H. Moghadam, S. Yoo, and F. Wu. Genetic Improvement for Adaptive Software Engineering. In G. Engels, editor, *SEAMS '14*, page Keynote, Hyderabad, India, 2014. ACM.

[23] M. Harman and B. F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, dec 2001.

[24] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo. Search Based Software Engineering: Techniques, Taxonomy, Tutorial. In *Empirical Software Engineering and Verification*, volume 7007 of *Lecture Notes in Computer Science*, pages 1–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[25] Z. A. Kocsis, G. Neumann, J. Swan, M. G. Epitropakis, A. E. I. Brownlee, S. O. Haraldsson, and E. Bowles. Repairing and Optimizing Hadoop hashCode Implementations. In *6th International Symposium, SSBSE 2014*, volume 8636 of *Lecture Notes in Computer Science*, pages 259–264. Springer Berlin Heidelberg, Fortaleza, Brazil, aug 2014.

[26] K. Lakhotia, M. Harman, and P. Mcminn. A Multi-objective Approach to Search-based Test Data Generation. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1098–1105, London, England, jul 2007. ACM.

[27] W. B. Langdon. Performance of genetic programming optimised Bowtie2 on genome comparison and analytic testing (GCAT) benchmarks. *BioData mining*, 8(1):1–7, 2015.

[28] W. B. Langdon and M. Harman. Genetically Improved CUDA C++ Software. In *Proceedings of the 17th European Conference on Genetic Programming, EuroGP 2014*, Lecture Notes in Computer Science, pages 1–12, Granada, Spain, 2014. Springer Berlin Heidelberg.

[29] W. B. Langdon and M. Harman. Grow and Graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 805–810, Madrid, Spain, jul 2015. ACM.

[30] W. B. Langdon and M. Harman. Optimising Existing Software with Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, feb 2015.

[31] W. B. Langdon, B. Y. H. Lam, J. Petke, and M. Harman. Improving CUDA DNA Analysis Software with Genetic Programming. In *GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, GECCO '15, pages 1063–1070, Madrid, jul 2015. ACM.

[32] W. B. Langdon, N. Veerapen, and G. Ochoa. Visualising the Search Landscape of the Triangle Program. In *EuroGP 2017*, pages 19–21, 2017.

[33] W. B. Langdon, A. Vilella, B. Y. H. Lam, J. Petke, and M. Harman. Benchmarking Genetically Improved BarraCUDA on Epigenetic Methylation NGS datasets and nVidia GPUs. In *GECCO 2016 Companion - Proceedings of the 2016 Genetic and Evolutionary Computation Conference*, pages 1131–1132, Denver, Colorado, USA, 2016. ACM.

[34] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13, Zurich, Swiss, jun 2012. IEEE.

[35] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[36] A. Marginean, E. T. Barr, M. Harman, and Y. Jia. Automated Transplantation of Call Graph and Layout Features into Kate. In *Search-Based Software Engineering*, volume 9275 of *Lecture Notes in Computer Science*, pages 262–268, Bergamo, Italy, aug 2015. Springer International Publishing.

[37] P. McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[38] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation*, To Appear, 2017.

[39] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class. In *17th European Conference on Genetic Programming, EuroGP 2014*, volume 8599 of *Lecture Notes in Computer Science*, pages 137–149, Granada, Spain, 2014. Springer Berlin Heidelberg.

[40] J. Petke, W. B. Langdon, and M. Harman. Applying Genetic Improvement to MiniSAT. In *5th International Symposium on Search-Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 257–262, St. Petersburg, Russia, aug 2013. Springer Berlin Heidelberg.

[41] S. Poulding and R. Feldt. Generating structured test data with specific properties using Nested Monte-Carlo Search. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1279—-1286, Vancouver, 2014. ACM.

[42] K. Salvesen, J. P. Galeotti, F. Gross, G. Fraser, and A. Zeller. Using Dynamic Symbolic Execution to Generate Inputs in Search-Based GUI Testing. *Proceedings - 8th International Workshop on Search-Based Software Testing, SBST 2015*, pages 32–35, 2015.

[43] K. Siggeirsdottir, U. Alfredsdottir, G. Einarsdottir, and B. Y. Jonsson. A new approach in vocational rehabilitation in Iceland: preliminary report. *Work*, 22(1):3–8, jan 2004.

[44] K. Siggeirsdottir, R. D. Brynjolfsdottir, S. O. Haraldsson, S. Vidar, E. G. Gudmundsson, J. H. Brynjolfsson, H. Jonsson, O. Hjaltason, and V. Gudnason. Determinants of outcome of vocational rehabilitation. *Work*, 55(3):577–583, nov 2016.

[45] F. C. M. Souza, M. Papadakis, Y. Le Traon, and M. E. Delamaro. Strong mutation-based test data generation using hill climbing. In *Proceedings of the 9th International Workshop on Search-Based Software Testing - SBST '16*, pages 45–54, Austin, Texas, 2016. ACM Press.

[46] J. Swan, M. G. Epitropakis, and J. R. Woodward. Gen-O-Fix: An embeddable framework for Dynamic Adaptive Genetic Improvement Programming. Technical Report CSM-195, Department of Computing Science and Mathematics University of Stirling, Stirling, UK, 2014.

[47] D. R. White. An Unsystematic Review of Genetic Improvement. In *45th CREST Open Workshop on Genetic Improvement*, London, 2016.

[48] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary Improvement of Programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, aug 2011.

[49] D. R. White, J. Clark, J. Jacob, and S. M. Poulding. Searching for resource-efficient programs. *Proceedings of the 10th annual conference on Genetic and evolutionary computation - GECCO '08*, (1):1775, 2008.

[50] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke. Deep Parameter Optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1375–1382, Madrid, Spain, jul 2015. ACM.

[51] K. Yeboah-Antwi and B. Baudry. Embedding Adaptivity in Software Systems using the. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 839–844, Madrid, Spain, 2015. ACM.