Scaling Genetic Improvement and Automated Program Repair

Mark Harman*
Meta Platforms Inc.
London, UK

ABSTRACT

This paper outlines techniques and research directions for scaling genetic improvement and automated program repair, highlighting possible directions for future work and open challenges.

KEYWORDS

Genetic Improvement; Automated Program Repair; Search Based Software Engineering (SBSE)

ACM Reference Format:
Mark Harman. 2022. Scaling Genetic Improvement and Automated Program Repair. In *International Workshop on Automated Program Repair (APR'22), May 19, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3524459.3527353

1 INTRODUCTION

Genetic Improvement [33] seeks to modify an existing software system to improve some aspect of its behaviour, without regressing on any existing desired behaviours. Genetic improvement is guided by software testing. The tests measure, not only the achievement of the improvement objective, but also the degree to which the improved program avoids regressions.

Automated genetic improvement technology has the ability to investigate many more candidate modifications than a human, but lacks the insight and context of a human software engineer. Nevertheless, as Section 2 outlines, the increasing scale of software systems tips the balance of opportunity away from human analysis in favour of automated techniques.

The increasing consumption of world energy resources in computation [13], with ever larger and more complex systems, creates many subtle dependencies between different levels of abstraction. A human software engineer, no matter how talented, cannot be expected to find all opportunities for optimising resource consumption across the full software stack, in the presence of such complexities. Automated genetic improvement can complement human decision-making and deployment choices, using computational search to explore the trade-off space, and reporting promising improvements to the human engineer.

*Mark Harman's scientific work is part supported by European Research Council (ERC), Advanced Fellowship grant number 741278; Evolutionary Program Improvement (EPIC) which is run out of University College London, where he is part time professor. He is a full time Research Scientist at Meta Platforms Inc.

APR'22, May 19, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9285-3/22/05…$15.00
https://doi.org/10.1145/3524459.3527353

Many software systems are increasingly containerised and deployed on cloud-based platforms [36]. Such cloud-based deployment facilitates testing across the full system stack, in safe sandbox isolation. As the technologies for cloud deployment become better developed, understood, and standardised, there will be increased opportunities for cloud deployment optimisation [20].

Search based program repair [14] can be thought of a special case of genetic improvement [33] where the aspect to be improved is a functional property of the system. Genetic improvement has also been applied to a wide variety of other non-functional properties, such as execution time, memory consumption and energy consumption [33]. Genetic improvement can also be thought of as a flavour of program synthesis. Unlike traditional synthesis [16, 29], the improvement technology is not merely constrained to guarantee meaning preserving transformations. Instead, it is free to investigate any modification that achieves a satisfactory signal that the improvement does not cause a regression. In this paper, we use genetic improvement to refer to search-based automated program repair, search based program synthesis, and any other technique that seeks to improve software guided by signal from software testing.

The ability to avoid regression depends crucially on the strength of the regression testing signal; poor quality tests will tend to lead to high probability of regression. Nevertheless, candidate improvements that perform well on the desired improvement criteria may give useful insights to software engineers, even when they do lead to regressions.

Furthermore, not all regressions are equal. There are situations in which some regressions can be tolerated when they also facilitate a sufficiently strong improvement in a desirable non-functional property. Software engineers make these engineering trade-offs all the time. Genetic improvement is a way to automate investigation of such multi criteria decision trade-off spaces [21]. Techniques that limit themselves only to known correctness–preserving transformations inherently cannot explore such engineering trade-off spaces.

Another exciting opportunity for genetic improvement is the way in which techniques can potentially be applied at scale. Search based program repair has already been deployed at scale, finding simple fixes in systems of tens of millions of lines of code [31]. However, this previous work merely fixed the most trivial of bugs, although arguably most widespread [24, 30]; the null pointer dereference.

In order to open the floodgates of opportunity to all of the many exciting developments from the research community, we need a concerted focus on *scalability* of genetic improvement. This paper attempts to make a contribution to this scalability agenda, by outlining current research that has potential to tackle some of these problems, avenues for deployment of scalable techniques, and open challenges for the research community.

2 SCALE AS OPPORTUNITY

In principle, genetic improvement can be applied at any scale, because it only seeks to make minor modifications to an existing (arbitrarily large) system rather than seeking to build systems from scratch. Therefore, the size of the code discovery' problem is largely driven by the size of the improvements made, not the size of the system to which the improvements are applied.

From an evolutionary computational perspective, and by analogy with natural evolution, the goal genetic improvement is to generate humans by 'improving' apes, rather than seeking to generate humans from scratch out of amino acids. The more significant is the scale of the software system, the more likely it is that there is robust test signal available.

As well as test cases written by the system's developers, there is production traffic, which can also be used to check for regressions. In this way, scalability is an *opportunity* as well as a challenge; the larger the system, the more comprehensive will be the likely available test signal to guide genetic improvement.

Also, at scale, the balance of expertise between genetic improvement and human domain knowledge shifts. Most developers can understand relatively small isolated components, and their domain knowledge helps them to identify optimisations that improve the software system. However, large-scale systems have grown so complex that few engineers understand the full stack operation; there are simply too many inter–operating services, backend systems, feature interactions, and subtle nuanced dependencies.

At this scale, the ability of an automated improvement technology to search large numbers of candidate improvements can considerably outperform human expertise. We have already seen these characteristics in the related problem of program transplantation. For transplantation, like genetic improvement, the complex context and dependencies involved lend themselves to an automated approach that is free to discard the vast majority of candidates it considers. This insight has led to award–winning human–competitive results for program transplantation using evolutionary computation [6].

Genetic improvement tends to tackle non-functional properties such as resource consumption and performance. Resource and performance bottlenecks in large-scale systems often arise as a result of complex interactions between multiple different system components. Human engineers are overwhelmed by complexity at this scale, naturally resorting to abstraction as the only way to cope. *Some* performance optimisations can be identified by understanding workflows at abstract levels. Nevertheless, many surprising performance improvements can be found only by identifying a set of detailed small-scale changes, which combine together, *across* levels of abstraction [27].

Of course, this does not mean that developers have no role to play in automated genetic improvement. Experience from industrial deployment of search based program repair [31] indicates that it is important that developers play the role of final gatekeeper. In a continuous integration system, changes need to be reviewed. Authorship and ownership are important engineering principles [1] that ensure accountability for code deployed to production. Therefore this final gatekeeper role is a key part of any deployment strategy.

In this industrial experience report [31], of all fixes found correct, approximately half were commandeered by the developer and re-written in an alternative style, even though they were semantically correct. Part of this behaviour might be attributable to understandable human hubris. However, not withstanding such 'human frailties', developers are the ones who undoubtedly know best how code should be represented in a system for on-going human maintenance. It is also the human engineers, not the automated improvement technology, that best understand the wider context in which the system is deployed.

3 LANDSCAPE OF SCALE OPPORTUNITIES FOR RESEARCHERS

When faced with the daunting challenges of tackling scale, researchers might be tempted to focus on perfection of techniques for smaller, more manageable, systems. However, the greatest impact on the software industry is most likely to be obtained using 'good enough' techniques applied at scale, rather than 'perfect' techniques applied only to simpler components. Fortunately, several existing avenues of current research will likely to prove important in achieving scalable genetic improvement. This section contains examples of such techniques. It also sets out challenges and open problems for the research community.

3.1 Test execution optimisation

One of the first scalability challenges tackled in the software engineering literature was that of optimising test execution. For some time it has been known that testing occupies a significant portion of the software engineering budget, arguably as much as half [32]. This observation motivated software engineering researchers to consider ways to optimise the selection, prioritisation and minimisation of test suites [11, 40].

Since genetic improvement is guided by testing, one of the key drivers of scalability is the ability to optimally deploy resources for software testing. Researchers have therefore considered ways to reuse, adapt and augment research associated with software test optimisation for genetic improvement [15].

Testing is likely to remain central to scalability, effectiveness and efficiency. Indeed, software testing is proving to be the success driver for some of the most challenging automated software development activities currently underway. For example testing is driving recent advances in automated code construction with AlphaCode [28] and language translation with TransCoder-ST [35].

Software testing is the oldest [37] the most widely practised method of ensuring software behaves as intended. When automated, it is essentially an automation of Popperian scientific investigation applied to software [23]. The quality of the genetic improvement is inherently dependent on the confidence the engineer can have in the signal from software testing. It is with genetic improvement that we find the strongest incarnation of the fundamental testing dilemma: testing can reveal the presence of faults, but can never demonstrate their absence [10]. Despite this apparently unassailable aphorism from Dijkstra, software testing has proved to be highly successful at giving confidence; software has become remarkably reliable, *despite* the obvious shortcomings of software testing [23].

Genetic improvement risks the strongest manifestation of testing's limitations, because of its high degree of automation. Genetic improvement will clearly, therefore, benefit from existing research on test optimisation. It may also provide fundamental contributions to this research agenda.

Testing's twin roles of identifying faults and giving confidence in correctness apply at the very heart of genetic improvement. Techniques that optimise the chance of finding faults soonest will help discard sub-optimal improvement candidates more quickly. Techniques that maximise the confidence in a candidate improvement, while minimising the effort required to provide that signal, will help guide the search towards promising candidates more quickly.

Research on scaling genetic improvement through test optimisation may also benefit the testing research agenda. The most pressing software testing questions engineers tend to ask can be more readily and thoroughly investigated through the lens of genetic improvement. Such questions include:

- 'When do I know that we have tested enough?'
- 'What confidence do I have that there are no serious bugs remaining?'
- 'What is the chance that this bug found by testing will manifest in production?'

If we are to deploy genetic improvement at scale, in practical software engineering scenarios, these questions are *unavoidable*. By contrast, for much of the wider testing research agenda, some or all of the three questions can be circumvented, despite being pressing practical concerns for practising software engineers.

3.2 The build effort problem

Large systems typically have build times that run into many minutes [3, 7, 22], even with smart infrastructure, bespoke build systems, and build caching. There has been a great deal of progress on build technology, giving rise to bespoke sophisticated build systems [12]. Nevertheless, build times for the largest systems have remained stubbornly in the region of 'multiple minutes' since the 1970s. Of course the sizes of the largest systems have grown by orders of magnitude since then, thereby ensuring that absolute build effort has remained high, despite advances in both build systems, and the hardware on which they execute.

This creates a 'built effort problem' for genetic improvement, because we need to evaluate many candidates to find an improved version, thereby rendering the technique inapplicable when the build effort is too large. The relative constancy of maximal build effort, over such a long period, suggests that the build effort problem will remain with us for the foreseeable future. Therefore, any technique that seeks to scale genetic improvement needs to operate effectively for systems with high build effort per build workflow execution. The remainder of this section considers ways in which we might tackle the build effort problem.

3.2.1 Exploiting techniques for mutation optimisation. Mutation testing has been a topic of study since the 1970s [9, 17]. It is an inherently computationally expensive testing technique, due to the large number of mutations from which the engineer can choose. This has made it the subject of research on mutation test optimisation. Much more work is needed to adapt these mutation testing orientated techniques to the closely related problem of genetic improvement.

Such work will not only help genetic improvement scale, but may yield insights into the fundamental nature of software faults, and the relationship between faults and fixes. That is, despite many years of study, the subtle interactions between syntax and semantics of software changes, faults and failures, and the interactions between multiple software changes remain comparatively poorly understood. As with software test optimisation, optimising mutation testing for genetic improvement may therefore provide a new avenue of investigation for these fundamental questions.

Program repair seeks to make a small change to remove a bug in an existing system. As such, it is the dual of mutation testing, which seeks to insert a simulated bug into an existing system. Both approaches make small syntactic changes to the software system, chosen from an enormous pool of potential candidates. This suggests that automated repair techniques may benefit from the extensive literature on mutation test optimisation.

Genetic improvement also makes small changes, but simply generalises from the problem of removing bugs, to that of improving the system in some way. Since this is a generalisation of search based program repair to arbitrary improvements, the same observations apply; genetic improvement, in general, can benefit from work on optimisation of selection, prioritisation, and minimisation of test mutants. Much previous work has focused on mutant selection, with approaches such as 'do faster', 'do smarter', 'do fewer'. A full treatment of all of these techniques is beyond the scope of the present paper. Fortunately, extensive surveys are available [26].

One technique associated with optimisation of mutation testing is the so-called 'metamutant' (aka 'mutant schemas' [38]); rather than compiling each mutant, the metamutant is a single compilation that encodes for multiple mutants. This is a technique that recognises the importance of minimising build effort. As the size of programs tackled by mutation testing increased, the community quickly realised that build effort started to dominate the computational cost of mutation testing. This observation led to mutation testing tools that optimised for minimisation of build effort [25].

Metamutants could also be adapted for genetic improvement. This would tackle the build problem by compiling into a single version of the program, multiple different possible improvements, each of which is selected at run time. However, this will not be applicable in every situation. For example, if the selection of an instance from a metamutant slows runtimes considerably, this may interfere with assessment of performance. Furthermore, the metamutant approach would not inapplicable when optimising for the footprint size of the software system to be improved. Fortunately, for problems like automated repair, the metamutant approach is highly applicable, and will facilitate significant scalability when the build effort problem is particularly pernicious.

3.2.2 Deep Parameter Exposure. Suppose we had a 'configuration' object that determined the behaviour of the property to be improved. With this object we could, at run time, simply supply different parameters to search the configuration space. Of course, most systems are not designed this way. Many of the key values that determine the non-functional properties to be improved are unlikely to be found in configuration files. Furthermore, some configuration files are not even run–time–loadable, but are compiled into the build of the system.

Nevertheless, suppose we choose to *redesign* the system under improvement to have a dynamically loaded configuration file that does, indeed, include those key values known to determine the operational characteristics to be improved. This run–time–configurable version of the system under improvement obviates the need to rebuild the system for every fitness evaluation. It allows us to genetically improve the software through parameter tuning, which is a well understood and widely studied problem.

The process of automatically identifying this dynamic configuration object is known as 'deep parameter optimisation' [39], not because it is based on deep machine learning (although such parameters might be identified using ML techniques), but rather because the parameters are *deeply* embedded in the software system. By exposing such deeply embedded parameters, they become a more readily available target for genetic improvement through parameter optimisation [33].

In the case of execution time optimisation, for example, such parameters might determine the value of data that flows into important loop control predicates, or other internal configuration values that influence the frequency or nature of hot path execution. Deep parameter optimisation offers promising ways to tackle the build effort problem for practical, scalable, genetic improvement and program repair.

3.3 The New Feature Build Cost Problem

Sometimes, engineers have the key insights into the candidate changes that need to be made to a system in order to improve behavioural properties of interest. For example, when seeking to improve systems to make them safer [2]. There are often many choices of feature, each with their own parameter spaces and characteristics. Many of these features will tend to interact with one another to produce emergent behaviours that are hard to predict, especially before any of the features has been built.

It is clearly wasteful to build non-trivial features into a system, only to discover that they interact in undesirable ways, or that parameter tunings cannot be found that would improve the behaviour of interest. Simulation can provide a viable alternative, in which the features are simulated, and the genetic improvement approach searches over the space of simulated features.

3.3.1 Simulation and Mechanism Design. At Meta, we have been experimenting with simulation–based approaches to tackling this problem. Such a simulation–based requirements analysis would have the potential to revolutionise the requirements process, with simulation dramatically scaling the potential of A/B testing, supporting counterfactual investigations, predictive feature deployment, and specialisation.

In particular, we used a mechanism design layer to simulate the effects of potential new features. We also simulated the effects due to different parameter tuning choices, including parameters pertinent to simulated features, through this mechanism layer [2]. Using the mechanism design approach we recognise that the space of optimisations to be found may involve non-trivial engineering effort. Therefore, rather than building each future in full, merely to discover that it has to be discarded as an inadequate source of improvement, we simulate the effect of a large space of potential features and their parameter tunings.

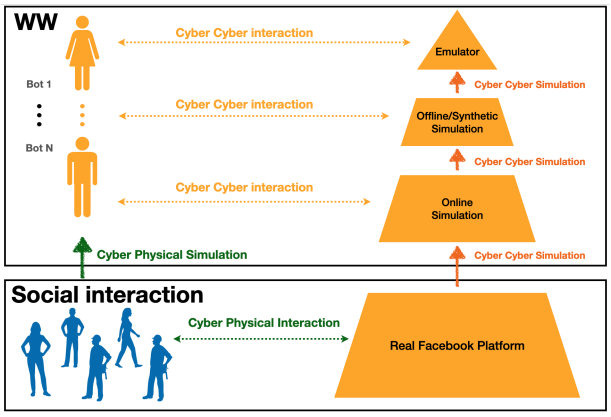

Figure 1: The WW Simulation Hierarchy: a recursive chain of cyber cyber digital twins. The figure is taken from the Meta WW team's EASE '21 keynote paper [4].

We use computational search to evaluate simulated feature spaces and optimisation choices. The search process identifies attractive candidates, which then feed into the decision process for requirements for software improvement by human engineers. The mechanism is a model of the execution behaviour of the candidate feature. A mechanism-based approach can be thought of, simultaneously, as an instance of a model based software engineering and as being an approach to requirements elicitation. We could also think of our approach as a form of search based software requirements elicitation [41], or as a form of simulated genetic improvement.

We conducted research and experimentation with this simulation–based approach, using mechanism design. The ultimately selected features still need to be built, potentially by human effort. However, the space of candidates we can prototype, and the complex interactions between their behaviours, can be explored more thoroughly, and at scale, through the simulation phase. In this way we ensure that precious human effort is reserved for only the most promising features, known to be likely to provide significant improvements.

In order to scale the approach, we built off-line versions of our simulation, which optimised the execution time for the simulation. These offline simulations are guided by online simulation. Online simulation executes on the real platform. Therefore its simulations, although more computationally expensive than their off-line counterparts, are highly realistic and faithful to the original system to be optimised. However, considering automated search is inherently computationally demanding, the off-line modes of simulation are needed to allow the engineer to negotiate the trade-off between faithfulness of simulation and execution time. This layered approach creates a hierarchical cyber cyber digital twin [4], as illustrated in Figure 1.

The hierarchy of digital twins supports the simulation of user behaviours on the real platform. The mechanism layer lies in between the bots that simulate user behaviours and the underlying system (which, in the hierarchy's base case, is the real software system, but can also be its offline digital twins). The approach, which we illustrate here in Figure 2, is covered in more detail in the WW team's RAISE 2021 keynote paper [5].

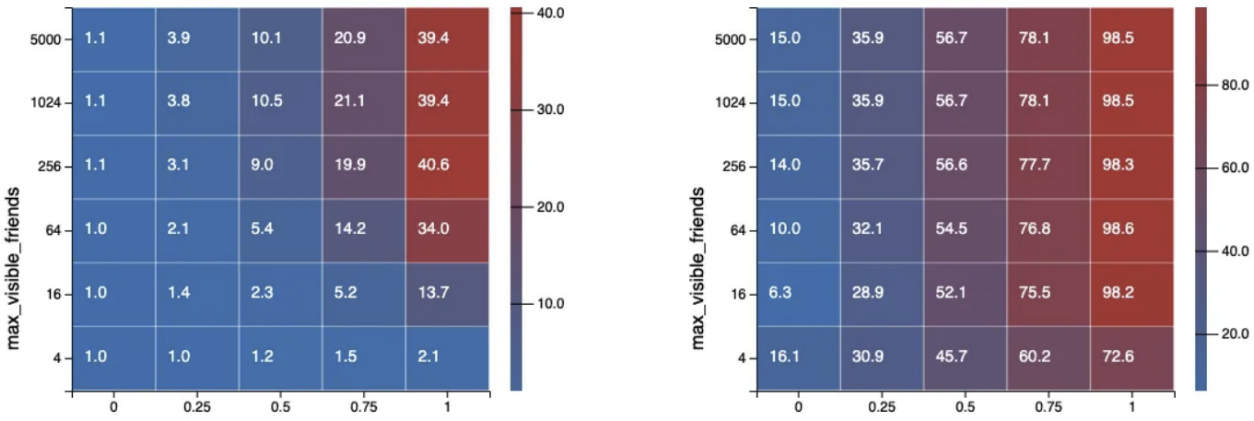

Figure 2: Heat maps from mechanism design exploration. Both heat maps show impact of different limits on maximum friends visible to a bot at each step (vertical axis) with different levels of propensity for the bot to engage in harmful behaviour (horizontal axis). These results were produced by the WW system, but the intervention in question is not in production and is shown here merely illustrative purposes only. The heat map on the left results from simulating on the high fidelity (but computationally expensive), online graph. The heat map on the right is result of simulating on a synthetic graph; it retains the overall distribution, thereby faithfully simulating the impact of the intervention, but it was computed approximately 100x faster. This figure is taken from the WW team's RAISE '21 keynote paper [5].

3.4 The Software Specialisation Problem

Automated software specialisation has long been a topic of interest in software engineering [8]. The key insight is that not all users' needs are the same; identifying different constituencies of users, and their corresponding use cases, opens up the potential for specialisation. Through specialisation, each user constituency runs a different version of the overall system. Sadly, the technical complexities of deploying different specialised versions, and the considerable effort required to disentangle these different versions, have made software specialisation highly challenging in practice. Automation is clearly key to tackling both problems. Genetic improvement provides a perfect fit, because different fitness functions automatically yield different versions of the system under improvement [21, 34].

Despite this, automated specialisation through genetic improvement, remains surprisingly underexplored. We need more work on genetic improvement for software product lines. There is a well-developed literature, and many practical industrial applications of software product lines, offering many opportunities for search based software engineering in general, and genetic improvement in particular [18].

We also need work on tailored deployment. The costs of maintaining different versions of software systems, and constructing them in the first place, currently combine to make specialisation unattractive. Much of the problem lies in the human-centric maintenance and construction costs. Genetic improvement can be used to explore potential candidate deployment opportunities, measuring and minimising the likely maintenance cost. We need more research on integrated end-to-end software specialisation that takes into account, not only the number of variants and their suitability, but also minimises the burden of onward maintenance and evolution.

3.5 The Hot Fix Problem

In order to deploy automated repair at scale, we need to be able to discover and deploy fixes in real time; the hot fix problem. There is considerable potential to tackle the hot fix problem using a combination of predictive modelling and genetic improvement. It is unrealistic to expect human engineers to cater for large numbers of possible changes in operational environments. Combining predictive modelling with genetic improvement we can automatically search the space of risks in an automated cost benefit analysis. In this way, we use predictive modelling to identify when a risk may be likely to materialise, and use genetic improvement to pre-search the space of potential hot fixes that can be deployed to ameliorate it affects. This combination provides one research avenue for automated autonomous adaptation through genetic improvement [19].

3.6 The Automated Acceptance Test Problem

The final arbiter of whether a genetically improved system can be deployed lies in compelling results from end–to–end testing. The more automated the approach, the more scalable it will be. However, this final arbiter has traditionally been considered to be a human, essentially playing the role of acceptance tester. For scalability, we need to minimise the reliance on human acceptance testing.

The problem is that the acceptance test needs to execute the entire system, in a realistic production setting. Traditional end–to–end tests often involve mocking details, and other compromises that mean that the test result may not be faithful to the real production setting, making acceptance tests unreliable. In theory this is not a problem: simply make a copy of the entire system, including all production databases, and execute this in a safe sandbox environment, in order to faithfully replicate likely behaviour of the improved system in production.

In practice, having such a safe sandbox for the copy of the entire system can be highly non-trivial. In industrial settings, fully end-to-end testing is surprisingly challenging; even tests designated as system tests typically involve *some* mocking of components, either for efficiency reasons, or to protect production; it is not always possible to execute a 'system' test on the real *system*. There is an unavoidable tension between the realism of an end–to–end system test (how *faithfully* does the test model what would happen in production on the real system?) and the risk *to* production of executing the test (what is the chance that running the test may, *itself*, adversely affect production behaviour of the system?).

For example, the test might affect state that is visible to some users, or pollute logs, intended to record behaviours only of real users. In theory, any unwanted state changes can either be rolled back or isolated, and any such pollution can be avoided, by ensuring separate logging streams. In practice, this may involve considerable engineering effort, especially when the system is not initially designed with such testability in mind.

Fortunately, the rapid uptake of cloud-based and containerised solutions offers an exciting avenue to full end–to–end testing, that faithfully mimics production, without necessarily risking any effect to production. Cloud-based deployment models lend themselves to optimisation [20] and have natural synergies with genetic improvement, offering platforms for evaluation and deployment of genetically improved software variants.

4 CONCLUSIONS: OPEN PROBLEMS AND SCALING CHALLENGES FOR RESEARCHERS

We conclude with six open problems for the research community, research on which may help facilitate genetic improvement scalability:

(1) How best can we reuse test optimisation research ?
(2) What techniques can be used to tackle the 'build effort problem' ?
(3) How can we develop approaches to the 'new feature problem', that best combine the complementary abilities of human and machine ?
(4) How can we best tackle the challenges of software specialisation using genetic improvement ?
(5) How can we develop integrated deployment techniques to tackle the hot fix problem ?
(6) How can we best achieve realism for 'automatic acceptance testing', without compromising production behaviours ?

5 ACKNOWLEDGEMENTS

The experience of working on the deployment simulation-based testing, Sapienz automated test design and Sapfix at Meta platforms Inc. has had a profound affect on my understanding of issues relating to scalable software engineering, testing, genetic improvement, and simulation.

I am deeply grateful to my many excellent colleagues at Meta for countless exciting conversations about these topics and their hard work and skill that has seen deployment and practical experience with related technologies.

REFERENCES

[1] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Shan He, Ralf Lämmel, Erik Meijer, Silvia Sapora, and Justin Spahr-Summers. 2020. Ownership at Large: Open Problems and Challenges in Ownership Management. *Proceedings of the 28th International Conference on Program Comprehension (ICPC 2020)* (2020).

[2] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Ralf Laemmel, Erik Meijer, Silvia Sapora, and Justin Spahr-Summers. 2020. WES: Agent-based User Interaction Simulation on Real Infrastructure. In *GI @ ICSE 2020*, Shin Yoo, Justyna Petke, Westley Weimer, and Bobby R. Bruce (Eds.). ACM, 276–284. https://doi.org/doi:10.1145/3387940.3392089 Invited Keynote.

[3] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Erik Meijer, Silvia Sapora, and Justin Spahr-Summers. 2021. Testing Web Enabled Simulation at Scale Using Metamorphic Testing. In *International Conference on Software Engineering (ICSE) Software Engineering in Practice (SEIP) track*. Virtual.

[4] John Ahlgren, Kinga Bojarczuk, Sophia Drossopoulou, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Simon Lucas, Erik Meijer, Steve Omohundro, Rubmary Rojas, Silvia Sapora, Jie M. Zhang, and Norm Zhou. 2021. Facebook's Cyber–Cyber and Cyber–Physical Digital Twins. In *25th International Conference on Evaluation and Assessment in Software Engineering (EASE 2021)*. Virtual.

[5] John Ahlgren, Kinga Bojarczuk, Inna Dvortsova, Mark Harman, Rayan Hatout, Maria Lomeli, Erik Meijer, and Silvia Sapora. 2021. Behavioural and Structural Imitation Models in Facebook's WW Simulation System (Keynote Paper). In 9^{th} *International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2021)*. virtual conference.

[6] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation (Gold Medal Winner at GECCO 2016 Human Competitive Results Competition — The HUMIES. Also winner of an ACM distinguished paper award at ISSTA 2015). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. 257–269.

[7] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 770–781.

[8] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-N. Volanschi. 1998. Tempo: specializing systems applications and beyond. *Comput. Surveys* 30, 3 (Sept. 1998). http://www.acm.org:80/pubs/citations/journals/surveys/1998-30-3es/a19-consel/ Article 19..

[9] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on test data selection: Help for the practical programmer. *IEEE Computer* 11 (1978), 31–41.

[10] Edsger W. Dijkstra. 1969. Structured programming. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF circulated privately.

[11] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A systematic review on regression test selection techniques. *Information & Software Technology* 52, 1 (2010), 14–30.

[12] Justin Etheredge. 2020. Why does it take so long to build software? https://www.simplethread.com/why-does-it-take-so-long-to-build-software/

[13] Erol Gelenbe and Yves Caseau. 2015. The impact of information technology on energy consumption and carbon emissions. *ubiquity* 2015, June (2015), 1–15.

[14] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[15] Giovani Guizzo, Justyna Petke, Federica Sarro, and Mark Harman. 2021. Enhancing genetic improvement of software with regression test selection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1323–1333.

[16] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119.

[17] R. G. Hamlet. 1977. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering* 3 (1977), 279–290.

[18] Mark Harman, Yue Jia, Jens Krinke, Bill Langdon, Justyna Petke, and Yuanyuan Zhang. 2014. Search based software engineering for software product line engineering: a survey and directions for future work (Keynote Paper). In 18^{th} *International Software Product Line Conference (SPLC 14)*. Florence, Italy, 5–18.

[19] Mark Harman, Yue Jia, William B. Langdon, Justyna Petke, Iman Hemati Moghadam, Shin Yoo, and Fan Wu. 2014. Genetic Improvement for Adaptive Software Engineering (Keynote Paper). In 9^{th} *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2014)* (Hyderabad, India). ACM, New York, NY, USA, 1–4. https://doi.org/10.1145/2593929.2600116

[20] Mark Harman, Kiran Lakhotia, Jeremy Singer, David White, and Shin Yoo. 2013. Cloud Engineering is Search Based Software Engineering Too. *Journal of Systems and Software* 86, 9 (2013), 2225–2241.

[21] Mark Harman, William B. Langdon, Yue Jia, David R. White, Andrea Arcuri, and John A. Clark. 2012. The GISMOE challenge: Constructing the Pareto Program Surface Using Genetic Programming to Find Better Programs (Keynote Paper).

In 27^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. Essen, Germany, 1–14.

[22] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 197–207.

[23] Charles Anthony Richard Hoare. 1996. How did software get so reliable without proof?. In *IEEE International Conference on Software Engineering (ICSE'96)*. IEEE Computer Society Press, Los Alamitos, California, USA. Keynote talk and extended abstract.

[24] Charles Anthony Richard Hoare. 2009. Null references: The Billion Dollar Mistake. In *QCON conference*. London, England. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

[25] Yue Jia and Mark Harman. 2008. Milu: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In 3^{rd} *Testing Academia and Industry Conference - Practice and Research Techniques (TAIC PART'08)*. Windsor, UK, 94–98.

[26] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (September–October 2011), 649 – 678.

[27] William B. Langdon and Mark Harman. 2014. Genetically Improved CUDA C++ Software. In 17^{th} *European Conference on Genetic Programming (EuroGP)*. Granada, Spain, 84–95.

[28] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. *Competition-Level Code Generation with AlphaCode*. Technical Report. DeepMind.

[29] Zohar Manna and Richard J. Waldinger. 1975. Knowledge and Reasoning in Program Synthesis. *Artificial Intelligence* 6, 2 (1975), 175–208.

[30] Ke Mao. 2017. *Multi-objective Search-based Mobile Testing*. Ph. D. Dissertation. University College London, Department of Computer Science, CREST centre.

[31] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. SapFix: Automated End-to-End Repair at Scale. In *International Conference on Software Engineering (ICSE) Software Engineering in Practice (SEIP) track*. Montreal, Canada.

[32] Glenford J. Myers. 1979. *The Art of Software Testing*. Wiley - Interscience, New York.

[33] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (June 2018), 415–432. https://doi.org/doi:10.1109/TEVC.2017.2693219

[34] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2014. Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class. In 17^{th} *European Conference on Genetic Programming (EuroGP)*. Granada, Spain, 132–143.

[35] Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging Automated Unit Tests for Unsupervised Code Translation. In 10^{th} *International Conference on Learning Representations (ICLR 2022)*. To appear.

[36] Salman Taherizadeh and Marko Grobelnik. 2020. Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications. *Advances in Engineering Software* 140 (2020), 102734.

[37] Alan M. Turing. 1949. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*. University Mathematical Laboratory, Cambridge, England, 67–69.

[38] Roland H Untch. 1995. *Schema-based mutation analysis: A new test data adequacy assessment method*. Ph. D. Dissertation. Clemson University.

[39] Fan Wu, Mark Harman, Yue Jia, Jens Krinke, and Westley Weimer. 2015. Deep Parameter Optimisation. In *Genetic and evolutionary computation conference (GECCO 2015)*. Madrid, Spain, 1375–1382.

[40] Shin Yoo and Mark Harman. 2012. Regression Testing Minimisation, Selection and Prioritisation: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[41] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. 2008. Search Based Requirements Optimisation: Existing Work and Challenges. In *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'08)*, Vol. 5025. Springer LNCS, Montpellier, France, 88–94.