Exterminating Bugs via Collective Information Recycling

George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

End-user software is executed billions of times daily, but the corresponding execution details ("by-products") are discarded. We hypothesize that, if suitably captured and aggregated, these by-products could substantially speed up the process of testing programs and proving them correct. Ironically, both testing and debugging involve simulating real-world conditions and executions, in essence trying to recreate in the lab some of these (previously available, but discarded) execution details.

This position paper proposes a way to recoup the execution information that is lost during everyday software use, aggregate it, and automatically turn it into bug fixes and proofs. The goal is to enable software to improve itself by "learning" from past failures and successes, leveraging the information-rich execution by-products that today are being wasted. We view *every* execution of a program as a test run and *aggregate* executions across the lifetime of a program into one gigantic test suite—i.e., we remove the distinction between software *use* and software *testing* and *verification*—with the purpose of substantially reducing software bug density.

1 Introduction

The quality of the software we use on a daily basis is poor, and this results from two seemingly insurmountable challenges: the *cognitive difficulty* of writing useful, complex software that is also provably correct, and the *computational cost* of verifying already written software. The cognitive aspect appears to be an inherent limitation of the human mind to formally reason about complex behaviors, such as those of software with millions of lines of code. The computational challenge faced by automated testing and proving techniques is due to the fact that the number of possible states and paths in a program is roughly exponential in the size of the program.

These two challenges—one cognitive and the other computational—make quality assurance expensive. Even though more than half of the resources in a typical development cycle are invested in testing and debugging [21], quality is still lacking: bugs manifest *after* software is shipped to users and persist long thereafter.

One could argue that we are losing the arms race against bugs. Empirical observation suggests the *density* of bugs in industrial-strength code has stayed relatively constant [21], yet the *volume* of code that goes into a general software product today has increased by many orders of magnitude (e.g., MS-DOS 1.0 had 4×10^3 lines of code in 1981, while the more recent Windows Vista has 5×10^7 lines of code [23]). This means that the number of overall bugs is growing at an alarming rate. If this trend continues, the lack of dependability in general-purpose end-user software could significantly affect a society that relies on software at every step: mobile phones, household appliances, entertainment, communication, office productivity, social networks, etc.

There exists a vast body of work aimed at improving software quality, and without this work one couldn't fathom building today's software systems. For example, some bugs can be eradicated using static analysis [10]. Alas, they often generate many false positives, potentially thousands for large-scale software, which then have to be inspected by human programmers [3]. Another approach to finding bugs is to use model checkers [25, 15]—they achieve high coverage and are sound, but suffer from poor scalability on large programs: the explored state space increases exponentially in the size of the program and the number of threads. Recently, several research projects have made progress in using symbolic execution to find bugs [12, 5]. Despite finding bugs with little human assistance, symbolic execution is still limited to small programs and cannot automatically synthesize fixes for the bugs it finds.

The cognitive challenge of software dependability can be overcome through automated learning, while the computational challenge can be overcome through massively distributed computation. To this end, we propose SoftBorg, a system that records execution information from programs running "in the wild" and automatically turns this information into fixes that make the programs more reliable over time, enabling software to improve itself in the same way intelligent beings accumulate experience and become more proficient with time. We advocate against the view that software *use* is separate from software *testing* or *verification*, and argue that these are in fact three naturally symbiotic facets of a program's life.

2 The Case for Information Recycling

In an abstract sense, the execution of a program *P* produces *information by-products*: information on which branches were taken, how many times each loop was executed, which locks were acquired and in what order, values returned by system calls, etc. Such information is key to verifying program properties—for example, traces of lock acquisitions/releases in a program's threads can be used to reason about the presence/absence of deadlocks.

Users execute software billions of times around the world—the equivalent of running billions of tests—yet, the corresponding execution by-products are discarded. If we could treat every end-user execution of a program as a test run, then the *aggregation* of all executions across the lifetime of a program (and across all copies of that program) would be equivalent to one big test suite. It is ironic that, when testing or debugging, developers try to simulate a tiny subset of possible executions, while that same information is being wasted at every program run. Such waste is even more absurd in light of the fact that no software organization can match the aggregate resources of a real user population—e.g., Microsoft Office runs on >500 million computers [22], which exceeds by orders of magnitude even the most optimistic estimates of how many servers are housed in Google's data centers [2].

In the battle against exponential bug proliferation, the aggregate hardware resources employed by end-users (laptops, tablets, smartphones, game consoles, desktops, servers, etc.) are more likely to be an effective weapon than the resources of any one organization. We postulate that the proper exploitation of end-user executions for testing and verification is key in arresting bug proliferation. However, this faces several challenges: capturing the by-products at low cost, collecting and analyzing them efficiently and securely, and turning them into practical fixes and proofs. Furthermore, these steps must be performed automatically, to enable software to *self*-improve, without relying on slow humans at every step.

To this end, we propose SoftBorg, a distributed platform that enables the various executing instances of an arbitrary program *P* to cooperate in (1) identifying the correct code in *P* and proving it correct, and (2) identifying the incorrect code and "correcting" it such that future failures are averted. When running on SoftBorg, the more a program is used, the more reliable it should become. We expect this can lead to orders-of-magnitude reduction in the bug density of popular software.

3 The SoftBorg Platform

SoftBorg is a collective of *pods* that lie underneath (ideally every) instance of a program *P* executing anywhere in the world. The pods observe *P*'s execution and relay the by-products over the Internet to the *hive*, a processing center (see Figure 1). The hive may be physically centralized (a cluster behind a web service), entirely distributed (running on end-users' machines), or hybrid.

The hive merges information extracted from by-products with its existing knowledge of *P*, identifies misbehaviors in *P*, synthesizes *fixes* that improve *P*, and distributes these fixes back to the pods, to be applied to their respective instances of *P*. For example, if *P* exhibits a deadlock pattern, SoftBorg can synthesize instrumentation that "protects" *P* from thread schedules that trigger that deadlock bug, thus avoiding future occurrences of that deadlock in *P* [16]. For correct behaviors, SoftBorg's hive produces and publishes *proofs* of *P*'s properties.

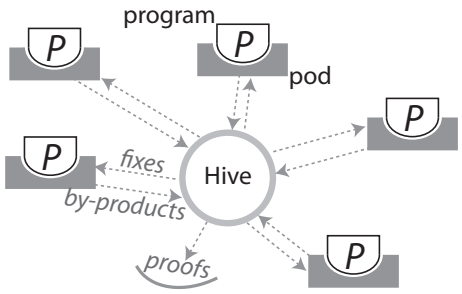

Figure 1. SoftBorg platform architecture

SoftBorg can also *guide* the execution of *P*'s instances to cover execution paths about which SoftBorg does not yet have sufficient information—for instance, it may guide *P* in exploring previously unseen thread schedules. Such steering augments the collective's knowledge derived from "naturally occurring" executions, accelerating the learning process. SoftBorg can also capture end-user feedback on program behavior, either directly or by inference (e.g., a user-terminated program was likely hung).

3.1 Capturing Execution By-Products

By-products that are of interest to SoftBorg fixes and proofs are primarily related to control flow, and they can be obtained by pods with a mix of dynamic and static instrumentation of *P*: aspect-oriented approaches [1], dynamic binary rewriting [19], and/or specialized runtimes [16]. To capture which branches were taken during an execution, we need one bit per branch—indicating whether the then or else branch was taken—which ends up encoding an execution as a bit-vector. The bit-vectors can be augmented with summaries of system call return values and thread schedules, as well as an indication of whether the execution was correct or not. The outcome of an execution is either determined by the pod explicitly (e.g., for crashes or deadlocks), or can reflect feedback provided by the end-user directly (e.g., via forceful program termination) or indirectly (e.g., an erratically jerked mouse suggests a program is being unusually slow).

The cost of capture can be reduced by focusing solely on branches that depend on program-external events; once they are fixed, the rest of the program execution is deterministic [8]. Modern CPUs also offer useful branch-counting facilities [14]. Sampling is effective too, especially if done in a coordinated fashion [18]: instead of

uniquely specifying a path, a recorded trace specifies a family of paths, but subsequent aggregation of traces can narrow down this family for the purpose of analysis.

An important question is how much and what types of information are contained in a trace. For example, traces might disclose private end-user information; even though initial ideas on anonymization exist [6], more study is needed. We are investigating ways to quantify this information content and develop a principled framework for reasoning about the balance between control flow details and privacy, as well as between recording granularity and runtime overhead—multicore CPUs offer ample opportunities for using parallelization to lower recording overhead, but the trade-offs are not yet well understood.

3.2 From By-Products to Collective Execution Trees

A program P is an encoding of a decision tree (Figure 2). To reason about P, the SoftBorg hive decodes this *execution tree* based on live executions. Each execution of P materializes one path from the root to a leaf. For large programs, the tree is overwhelmingly large: its size is roughly exponential in the number of branches. Different thread interleavings further compound this challenge, because they weave different executions out of otherwise identical thread-level execution paths.

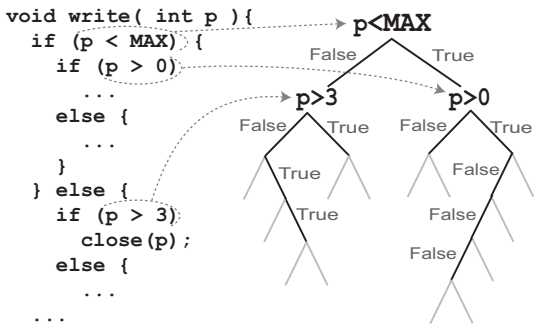

Figure 2. Every program encodes an execution tree

Statically constructing the execution tree—as done in classic symbolic execution [17, 5]—is prohibitively expensive for large programs. One reason is that, at each branching point, one must decide which branch(es) is/are feasible, in order to guarantee that there indeed exists a combination of program inputs and thread schedule that would produce that path in a live execution. Deciding feasibility requires solving complex constraints on variables in P and its environment, amounting to deciding propositional satisfiability—an NP-complete problem.

Instead, SoftBorg builds P's execution tree *dynamically* by aggregating execution paths that result from end-users' "natural" executions of P and transmitted to the hive by the pods. Given that each path occurred during an actual execution, it is guaranteed feasible, so runtime constraint solving is not necessary. Execution traces are encoded as bit-vectors indicating the direction of input-dependent branches; merging a path into an existing (incomplete) execution tree consists of reconstructing the deterministic branches, identifying a lowest common ancestor, and pasting the path into the tree (Figure 3).

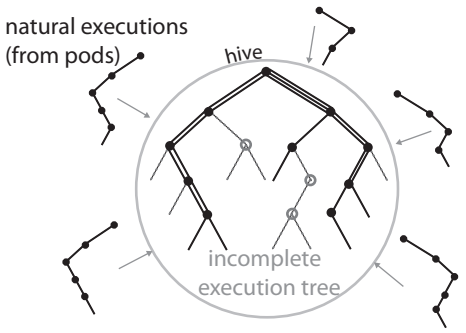

Figure 3. Naturally occurring execution paths are collected and dynamically merged into an execution tree

3.3 From Execution Trees to Proofs and Fixes

A complete exploration of all paths leads to a proof, while a test is just a weaker proof that covers a smaller subset of the paths. In SoftBorg, we unify tests and proofs along a single spectrum: a *cooperative prover* combines aggregations of individual test cases (i.e., naturally occurring executions) with additional symbolic program analysis (performed by the hive) to synthesize proofs or synthesize the bug fixes required to enable these proofs.

As the hive receives execution traces, it continuously reasons about the program and attempts to prove useful properties about P, thus incrementally assembling *cumulative proofs* of correctness. If the tree (or part of it) is found to be complete, the hive produces a proof that the program (or part of it) is guaranteed to satisfy the desired property. This proof is then published.

There are two hurdles to building such a proof: (1) program bugs, i.e., counter-examples that invalidate the attempted proof, and (2) execution subtrees that do not get naturally explored in a reasonable amount of time.

The first hurdle—bugs—are program behaviors that must be corrected in order to make the proof possible. For instance, the tree may have a path for which deadlock can occur. In this case, SoftBorg uses a runtime-based mechanism or minor instrumentation to modify P such that its threads avoid the conditions under which that deadlock occurs, helping P avert the deadlock in future executions [16]. SoftBorg can also employ other approaches to synthesizing bug fixes [24]. By correcting P's behavior in this way, SoftBorg "smoothes over" the hurdles that prevent the proof. We are studying generalized techniques for on-the-fly behavior fixes that can generate an enhanced version of P for which SoftBorg can provide behavioral guarantees. Since it is not yet clear how many types of bugs can be fixed automatically, we also provision for a repair lab that suggests plausible fixes to developers, who then manually choose the correct one.

The second hurdle—an incomplete tree—is encountered when, even after many executions, the aggregate tree still has unexplored paths. Instead of waiting for the tree to become complete, SoftBorg uses symbolic analysis of the program to (1) reason about the incomplete tree, and (2) identify directions toward which to guide the pods to "fill in" the gaps. For example, there may be certain thread interleavings that are rare in practice, but might be hiding bugs—SoftBorg instructs some of the pods to guide their program copies toward those thread schedules. None of the execution guidance ever modifies *P*'s semantics, because SoftBorg will only guide *P* down feasible paths through its execution tree. SoftBorg can also produce specific test cases to guide execution, stated in terms of inputs or in terms of system call faults to be injected (e.g., a short socket `read()`). Execution guidance enables accelerated learning: the SoftBorg collective obtains the missing traces more rapidly than if it waited for the executions to occur naturally, thus learning sooner about *P*'s behavior in rare corner cases.

Since SoftBorg must work for real-world software, it must reason automatically about large execution trees: it needs to decide which pieces are missing from its proof puzzle, needs to determine what instrumentation will fix program behavior, must reason about whether this instrumentation could affect *P* in undesired ways, and needs to decide whether the instrumentation invalidates the hive's existing knowledge and proofs. Therefore, SoftBorg must perform program analysis at an unprecedented scale and overcome exponential path explosion.

4 Cooperative Symbolic Execution

For such large-scale analyses, we need to harness as much of the unused end-user computing power as possible. We therefore *parallelize* symbolic execution using the Cloud9 framework [4] and *distribute* the analysis of the execution tree to the hive's nodes (which could include as many as all machines running SoftBorg).

A key question is how to distribute subtrees of the execution tree among the hive's nodes, which are mostly end-user machines communicating over a potentially unreliable network? One way to do this is to statically split the execution tree and farm off subtrees to worker nodes. Unfortunately, the contents and shape of the execution tree remain *unknown* until the tree is actually explored, and thus finding an appropriate partition is undecidable.

Instead, SoftBorg partitions the execution tree *dynamically*, as it is being explored, and the hive nodes exchange information on what they have found thus far.

In order to cope with the unpredictability of the subtrees, we build upon modern portfolio theory [20]. When investing in financial instruments, choosing the equities with the highest return is "undecidable", so one must invest in parallel in several equities, in order to balance the risk/reward mix. In SoftBorg, equities correspond to roots of subtrees in the execution tree, and the capital invested in each equity corresponds to the hive nodes allocated to analyze them. Portfolio theory offers strategies that appear promising for our problem, such as diversification, speculation, and efficient frontier. Preliminary results suggest that the portfolio approach works for constraint solving: by replacing a single SAT solver with a portfolio of three different SAT solvers running in parallel, we achieved a $10\times$ speedup in constraint solving time with only a $3\times$ increase in computation resources. We believe that each solver is fast in solving some path constraints but slow on others and, for most constraints, at least one solver completes much faster than the others.

A second, complementary way for the SoftBorg hive to scale symbolic analysis is through controlled relaxation of analysis precision. The traditional assumption about the execution of a system is that the state at any point in time is consistent, i.e., there exists a feasible execution path from the start state to the current state. However, there are many analyses for which this assumption is unnecessarily strong, and the cost of providing such consistency during symbolic execution is often prohibitively high. For example, when doing unit testing, one typically exercises the unit in ways that are consistent with the unit's interface, regardless of whether all those paths are indeed feasible in an integrated system. This overapproximates the paths through the unit, but reasoning at the unit level (instead of system level) can be faster despite the fact that overapproximation introduces more paths to analyze. If the unit behaves correctly for a superset of the feasible paths, then it is guaranteed to behave correctly for all feasible paths.

SoftBorg relies on S^2E [7] for such analyses. Relaxed execution consistency enables *in-vivo program analysis*, a way to study a program's behavior along all paths inside its real environment (libraries, kernel, drivers, etc.) *without* the use of abstract models. This approach automatically slices the exploration of paths in large software into thorough exploration of the code of interest plus quick single-path exploration of the rest. More specifically, the code of interest lies inside a "symbolic domain", where SoftBorg can reason symbolically about the execution paths, whereas the rest of the system remains outside the domain. As execution weaves its way between the symbolic and non-symbolic domains, S^2E converts on-the-fly the representation of program state between symbolic and normal (concrete) state. These conversions are precise, and they are governed by the rules of the chosen execution consistency level [7]. As an added benefit, S^2E can operate directly on binaries, thus allowing SoftBorg to accommodate proprietary software.

SoftBorg is not yet a completely built and validated system. In this position paper we glossed over many details and potential difficulties, but we recognize that many hurdles do exist. It is our hypothesis that the combination of techniques described here does have the potential to reduce bug density by an order of magnitude or more.

5 Related Work

SoftBorg is a logical descendant of today's generation of error reporting mechanisms, such as WER [11]. SoftBorg proposes a few new steps: (1) a recognition of the high information content of execution by-products and a systematic way to collect and reason about them; (2) cumulative proofs that use natural program executions to incrementally construct proofs; (3) closure of the quality feedback loop by automatically producing and distributing fixes, thus minimizing reliance on slow humans; and (4) cooperative symbolic execution.

The construction of bigger and more concurrent programs offers bugs more opportunities to hide and makes them ever more difficult to fix; gaining confidence in a program's correctness is getting harder. Many techniques have been advocated for improving software reliability, ranging from better software engineering and object oriented programming to formal methods. The approach proposed here complements these techniques, providing an additional, orthogonal way to combat bugs. Furthermore, SoftBorg is a good match for the chaotic way in which end-user software is being developed today.

The cooperative aspect of SoftBorg is inspired in part by the cooperative bug isolation project [18], which uses a sampling infrastructure for gathering information from the executions experienced by a program's user community. That project transforms assertion-dense code into code with fewer assertions distributed randomly among the different copies of running programs; as these copies run in the field, triggered assertions are reported centrally and interpolated such that the location of bugs is inferred. Cooperative bug isolation does not diagnose bugs nor generate proofs or hints for fixing the bugs.

Record-replay systems [9, 13] collect execution information with the goal of replaying an execution. Systems like R2 [13] reduce recording overhead by asking developers to specify the interfaces where to capture the program's interactions. Castro et al. [6] discuss the tradeoff between recording overhead and post-factum analysis; they are also among the first to seriously analyze the privacy implications of execution recording. SoftBorg's recording techniques build upon these previous systems.

6 Conclusion

In this paper, we proposed a solution to an acute practical problem: the alarmingly poor quality of everyday end-user software. We advocated merging *regular use* of software with *self-testing*, *self-fixing*, and *self-verification*. The SoftBorg platform combines runtime recording with online, distributed construction of execution trees and with automated derivation of proofs and fixes. To reason about large-scale programs, we propose cooperative symbolic execution, which harnesses end-users' machines to collectively solve analysis problems that are beyond the reach of any single organization. We view SoftBorg as a new step toward the (potentially Utopian) goal of zero-defect real-world software.

References

[1] AspectJ. http://www.eclipse.org/aspectj.

[2] B. Barrett. Google's insane number of servers visualized. http://gizmodo.com/#!5517041/googles-insane-number-of-servers-visualized, 2010.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2), 2010.

[4] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *ACM EuroSys European Conf. on Computer Systems*, 2011.

[5] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[6] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[7] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[8] O. Crameri, R. Bianchini, and W. Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *ACM EuroSys European Conf. on Computer Systems*, 2011.

[9] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Symp. on Operating Sys. Design and Implem.*, 2002.

[10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Conf. on Programming Language Design and Implem.*, 2002.

[11] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Symp. on Operating Systems Principles*, 2009.

[12] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conf. on Programming Language Design and Implem.*, 2005.

[13] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[14] Intel processor events. http://software.intel.com/sites/products/documentation/hpc/amplifierxe/en-us/lin/ug_docs/reference, 2011.

[15] Java PathFinder. http://javapathfinder.sourceforge.net, 2007.

[16] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[17] J. C. King. A new approach to program testing. In *Intl. Conf. on Reliable Software*, 1975.

[18] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Conf. on Programming Language Design and Implem.*, 2005.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *Conf. on Programming Language Design and Implem.*, 2005.

[20] H. M. Markowitz. *Portfolio Selection: Efficient Diversification of Investments*. John Wiley & Sons, Inc., New York, 1959.

[21] S. McConnell. *Code Complete*. Microsoft Press, 2004.

[22] Microsoft. Office 2010. http://www.microsoft.com/business/smb/en-hk/office/office-2010.mspx, 2009.

[23] The operating system documentation project. http://www.operating-system.org, 2010.

[24] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Symp. on Operating Systems Principles*, 2009.

[25] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *First IEEE Symp. on Logic in Computer Science*, 1986.