Object-Level Recombination of Commodity Applications

Blair Foster
School of Computer Science
Carleton University
Ottawa, ON Canada K1S 5B6
blair.foster@gmail.com

Anil Somayaji
School of Computer Science
Carleton University
Ottawa, ON Canada K1S 5B6
soma@scs.carleton.ca

ABSTRACT

This paper presents ObjRecombGA, a genetic algorithm framework for recombining related programs at the object file level. A genetic algorithm guides the selection of object files, while a robust link resolver allows working program binaries to be produced from the object files derived from two ancestor programs. Tests on compiled C programs, including a simple web browser and a well-known 3D video game, show that functional program variants can be created that exhibit key features of both ancestor programs. This work illustrates the feasibility of applying evolutionary techniques directly to commodity applications.

Categories and Subject Descriptors

I.2.2 [**Computing Methodologies**]: Artificial Intelligence—*Automatic Programming, Program synthesis*

General Terms

Experimentation

Keywords

genetic algorithms, genetic programming, software recombination, ObjRecombGA, object-level recombination, commodity programs

1. INTRODUCTION

The dream of evolving computer programs has long been pursued by those interested in evolutionary computation. Building on the work of Koza[9], we now have techniques for evolving programs to solve many relatively complex algorithmic problems and for evolving solutions not just in LISP but in standard programming languages such as C [10]. Evolved code, however, exists in a separate universe from the bulk of computer code, except when connected via careful encapsulation or manual coding. While existing programs can exhibit many lifelike properties when viewed as part of complex software ecosystems [1], the reality is that most programs only evolve through manual programmer intervention.

Here we argue that it is feasible to evolve commodity programs automatically, and specifically that such evolutionary processes can produce derivative, functional programs with potentially new feature combinations. The key insight is that the fundamental operator of genetic algorithms, recombination, can work on pairs of conventional programs *so long as those programs are closely enough related.* Much as a dog and a camel cannot mate successfully, we cannot expect recombination between a web browser and a word processor to produce viable offspring. However, if we instead recombine different versions of "the same" program—ones that share a development history but whose code diverged relatively recently—we can get new functional variants.

To demonstrate the feasibility of this approach, we have developed ObjRecombGA, a genetic algorithm framework for recombining the object files of standard Linux binary programs. Given two ancestor programs, ObjRecombGA evolves populations of bit strings that represent recombinations of their respective object files. ObjRecombGA handles minor linking problems, screens binaries for basic functionality automatically, and calculates program fitness using user-defined shell scripts. At the end of a run, the user is presented with a set of mostly functional programs that can be manually or automatically inspected further for desired properties. To evaluate the effectiveness of our framework, we recombined versions of a simple UNIX command line program GNU `sed` [8], a simple graphical web browser Dillo [12], and a popular 3D game Quake [14] and were able to create very functional variants. In particular, we successfully evolved functional variants of Quake that combined the desirable qualities (e.g., user interface and general I/O features) of divergent versions, created by different independent development teams, into one executable.

Because of the behavioral complexity of non-trivial programs, we have not been able to develop fitness functions that completely evaluate the functionality of program variants while allowing the variation that makes program recombination interesting. Thus, even though ObjRecombGA is structured as a traditional GA, in practice the evolutionary process it enables resembles the selective breeding of domestic animals or Dawkins's "blind watchmaker" [5]—at the end of a run, programs need to be manually evaluated to determine their suitability for use or for further recombination. Even with the manual aspects of the current process, however, ObjRecombGA offers a way to create "mashups" of

GECCO'10, July 7–11, 2010, Portland, Oregon, USA.

variants of standard applications without the need for any manual programming. As such, we believe ObjRecombGA represents a small but significant step toward the evolution of software in common practice.

The rest of this paper proceeds as follows. First, we explain our strategy for recombining ancestor programs in Section 2 and discuss details of our implementation in Section 3. We then describe the results of recombining existing programs in Section 4. We describe related work evolving programs in Section 5. Section 6 concludes.

2. DESIGN

ObjRecombGA is a traditional genetic algorithm in many respects. It operates on a linear bit-string chromosome, populations are initialized with random bit strings, it uses one or two-point crossover, and it performs tournament selection with elitism. The key difference is that ObjRecombGA's chromosomes encode program variants in the form of collections of object files.

Object files are the standard compilation unit for languages that compile directly into machine language. For example, with C language programs in a standard UNIX-like environment, program source code is divided into multiple files, each of which is given a `.c` extension. These files are converted into an executable by first compiling each into `.o` files containing a machine language translation of the C source code. These `.o` files cannot be executed directly, however, because they have external code and data references (symbols). To make an executable, then, multiple `.o` files are combined (linked) such that all external references are resolved appropriately.[1]

To evaluate the fitness of a bit string, ObjRecombGA selects object files from one of two ancestor programs that are selected by the user at the start of the run. Each position in the bit string refers to a unique object filename; 0 refers to the first ancestor, 1 refers to the other. The selected object files are linked to form a complete program binary. Then, the child binary is executed under the control of a user-supplied script to determine its fitness.

Note that this process, if applied naively, is doomed: arbitrary selections of object files taken from two different programs will inevitably have unresolved symbols—the program won't even link, let alone execute. If we limit ourselves to closely related programs, i.e., programs that are based on the same original code base, the situation is much better, as they are likely to both have object files with the same names and approximately the same functionality and layout; nevertheless problems are still quite likely.

To see why, consider a simple a program consisting of only two object files, A and B. In version *alpha* (α) of the program, object A contains two functions, fAn and fAm, along with two global data items, dAx and dAy. Object B also has two functions, fBn and fBm, in addition to a single global data item, dBx. In version *beta* (β), object A has been modified through the removal of function fAm and the substitution of data item dAy with dAz. Figure 1 illustrates objects A and B from both versions with arrows indicating external references between the two objects.

[1]Note that libraries are just indexed collections of `.o` files. Dynamic libraries are collections of object files that are linked with an executable at runtime, rather than when the executable is first built.

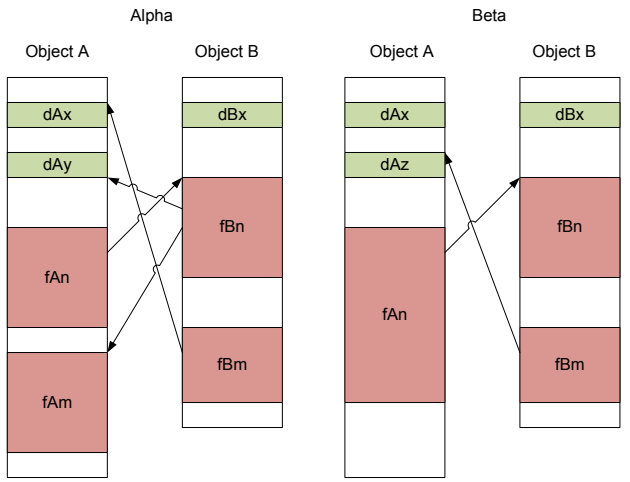


Figure 1: Object Files from versions *Alpha* and *Beta*

Attempting to recombine and link object A_α with B_β will be problematic because function fBm from object B_β has an unsatisfied reference to a global data item dAz, which is not present in object A_α. Likewise, attempting to recombine and link object B_α with object A_β is also problematic because of the lack of fAm.

The key to resolving this difficulty is to allow redundant copies of object files, i.e., allow both A_α and A_β to be used when creating an offspring. While the object file selected by the GA will take precedence, any unresolved references are satisfied using the object file from the other parent. (If only one parent has a given object file, that file will always be used.) Note that ObjRecombGA only works with two ancestor programs; there are always at most two versions of every object file. Thus, we can view ObjRecombGA's individuals as specifying linking precedence rather than explicit inclusion or exclusion—all ancestor object files are potentially available to every individual in the population.

ObjRecombGA allows user-defined fitness functions in the form of standard `bash` shell scripts; however, before such scripts are executed, individuals are first evaluated with two simple tests to check first whether an executable was successfully created, and second checks that the resultant executable runs without crashing. The score of these preliminary tests is added to the user-defined fitness function; thus, supplied scripts just need to score a program's functionality.

3. IMPLEMENTATION DETAILS

Having given an overview of ObjRecombGA in the previous section, we now discuss several implementation details. As can be seen in Figure 2, ObjRecombGA presents the user with a simple graphical frontend for setting the inputs of a new run. Many of the general settings, such as population size and number of generations, are standard genetic algorithm settings. Others, however, are unique to ObjRecombGA. In particular, the user must tell ObjRecombGA where to find the object files for the two ancestors (the primary and secondary directories) and must tell it how to build and test executables. Additional options include the ability to provide an initial population and the ability to restrict which common object files should be subject to the recombination

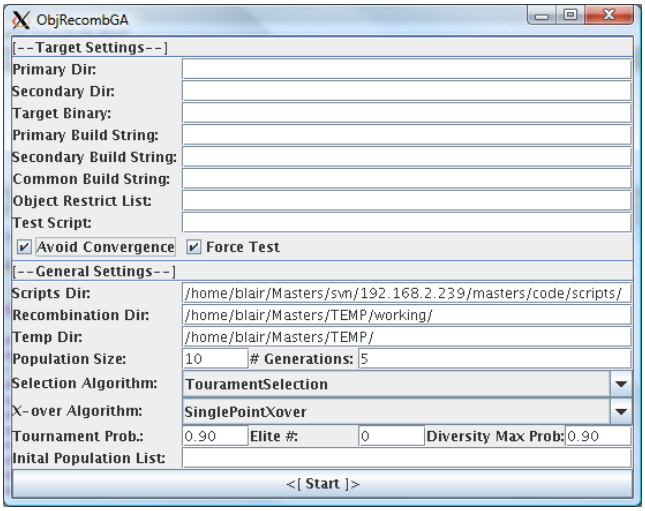


Figure 2: The ObjRecombGA interface.

process. These options enable users to target a particular set of object files and specify exact recombinations of those object files.

Note that ObjRecombGA does not use the primary and secondary build strings as-is; the purpose of these strings, rather, is to identify all relevant object files so they may be analyzed and manipulated. Other than system libraries (which are included as-is in the final link stage), ObjRecombGA divides object files into three categories:

1. Object files unique to one ancestor.
2. Object files shared between the ancestors (that are bitwise identical).
3. Object files that have the same name but have differing contents between the two ancestors.

Object files in the first two categories are put into a single, per variant, runtime-created library that is linked against; the object files in category three are the ones represented by bits in ObjRecombGA's chromosomes.

Note that because we are combining code from two very similar yet different ancestor programs, we cannot use a standard linker—there are too many duplicated symbols. Even worse, we must bring in code from non-selected object files: while our bit string may say to use `foo.o` from ancestor A, we may have references that can only be resolved using `foo.o` from ancestor B.

We solve this problem in two steps. First, at the start of a run, all object files and libraries are enumerated and their imported and exported symbols are cataloged. Imports which resolve to symbols not defined inside the ancestor directories are purged, as they are assumed to refer to standard external libraries that will be supplied to the final linker command. This inventory allows all duplicated symbols to be identified as well as all dependencies across object files within each version.

Then, to generate a program variant, ObjRecombGA uses a link resolver that inspects and rewrites (copies of) the ancestors' object files in order to make them refer to the preferred definition of each symbol. Further, it follows dependency chains to make sure the code and data so referred have what *they* need included in the final binary as well—all while respecting the object file preferences expressed in a given individual's chromosome. The modified object files are then passed on to the standard GNU `ld` linker to create a functional program.

The link resolver has been the key technical challenge in developing ObjRecombGA. While its functionality is relatively straightforward, the complexity of object file formats necessitates a relatively convoluted implementation. To facilitate the research of others, we are planning on releasing a technical report or open source implementation of ObjRecombGA's link resolver in the near future.

While the link resolver will generally produce an executable, occasionally the link fails due to limitations of our current link resolver (although this has become increasingly rare as our implementation has matured). Further, many executables so produced will hardly run: they crash on startup or shortly thereafter. Rather than attempt to fix these issues programmatically, we instead rely upon the genetic algorithm framework to search for functional programs.

Specifically, ObjRecombGA's has a built-in fitness function that scores each individual on whether it did build successfully and does it run without generating a crash dump. A successfully compiled program gets 1 added to its fitness score; a program that also runs without crashing (creating a core dump file) gets another 1 added to its score. The user-supplied fitness shell script is then run with the generated program as an argument; these scripts return an integral value that is added again to the program's fitness score to produce a final value.

Currently, ObjRecombGA only supports object files from compiled C programs; however, extending this to support C++ would require minimal effort as the two languages share a very similar linking stage.

4. RESULTS

To evaluate ObjRecombGA's ability to recombine different versions of a given type of program, we ran it on versions of three programs:[2]

- GNU `sed` [8], a version of the standard UNIX command line stream editor,
- Dillo [12], a small graphical web browser, and
- Quake [14], the popular first-person shooter originally developed by ID Software.

We first looked at `sed` because it was a relatively simple yet non-trivial program that we could easily inspect and evaluate its fitness. We then studied Dillo as an example of a graphical application of moderate side but containing complex internal algorithm (specifically, its rendering code). In both these cases, we evaluated to what extent older and newer program versions could be successfully recombined into (mostly) functional variants. This work can be seen as a simulation of patching older program versions with fixes present in newer ones.

[2]All three of these programs are licensed under versions of the Free Software Foundation's GNU General Public License (GPL), and as such permit the creation and distribution of modified binary versions so long as the modified source code is included.

Version Pair	Files	Unique	Unstable
4.0.6 × 3.02	4	12	9
4.0.9 × 3.02	4	12	19
4.0.9 × 4.0.6	5	21	0
4.1.2 × 4.0.6	5	21	15
4.1.3 × 4.0.7	5	15	4
4.1.4 × 4.0.8	5	20	10
4.1.5 × 3.01	4	14	12
4.1.5 × 4.1.1	7	21	0

Table 1: Results for the tested GNU `sed` ancestor pairs. "Files" refers to the number of files available for recombination. "Unique" is the number of unique variants generated; 68 variants were created for each pair. "Unstable" is the count of variants that crashed upon execution.

After our success with `sed` and Dillo, we chose to study a software ecosystem that had a more diverse population. Many programmers have created Quake variants. Some authors have focused on bug fixes and code cleanups; others have implemented a wide variety of gameplay, appearance, and interface changes. The individuality of the Quake variants is remarkable for the degree to which it resembles the diversity of individuals belonging to the same biological species. Our tests with Quake, then, were more ambitious: we wished to see if the key features of different branches could be brought together in a single binary using ObjRecombGA.

Note that we did not run ObjRecombGA with an aim to find an optimal cross; indeed, such an optimum was not well defined in these runs because so many individuals were able to achieve maximum fitness. We made this choice because our aim was to explore the characteristics of recombined programs, not to find the best one (whatever that might be). It is possible that other search methods might produce better results. What is significant here is that ObjRecombGA found programs that were of sufficient quality to warrant manual inspection. Specifically, we found a number of interesting variations in program behavior, ones that reveal the potential of object-level recombination to produce interesting, perhaps even useful levels of software diversity.

The results of our tests are detailed below.

4.1 GNU `sed`

For GNU `sed`, we selected eight pairs of versions as detailed in Table 1 and ran ObjRecombGA on them with a fitness function based upon success in running six scripts from the Sed Script Archive [4], with one point given for each line of output that matched that of a standard version of `sed` on the same scripts. We used a size 12 population, size 2 tournaments with 90% selection probability, and an elitism size of 4 over 8 generations. Over this relatively short run, as expected we saw no convergence toward all 0's or 1's; thus, all tested individuals were mixes of code from both ancestors.

Note that there were, at most, only seven object files available for recombination; thus, excluding the original ancestors, the number of potential variants ranged from 14 (2^4-2) to 126 (2^7-2)—very small search spaces. Indeed, we often created more individuals than there were possible combinations. (As GNU `sed` was the equivalent of max-ones for ObjRecombGA, we did not consider the redundant individuals to be significant.) All created variants were able to be linked successfully.

Version Pair	Files	Unique	Unstable
0.8.5 × 0.8.2	56	34	0
0.8.5 × 0.8.0	52	262	86
0.8.5 × 0.7.3	51	233	165
0.8.4 × 0.8.1	52	274	145
0.8.3 × 0.8.0	52	227	59
0.8.0 × 0.7.3	51	348	91

Table 2: Results for the tested Dillo ancestor pairs. 448 individuals were generated for each pair.

Interestingly, the cross with the largest search space, 4.1.1 × 4.1.5, had the most viable programs (no crashes); this level of success, however, is probably due to the close version numbers of the programs. In contrast, some crosses, such as 4.0.9 × 3.02, produced mostly non-viable programs. All combinations, however, were able to produce some variants that contained code from both ancestors.

The key result from these tests was that object-level recombination seemed to work—we were able to create generally working variants even between highly divergent ancestor programs.

4.2 Dillo

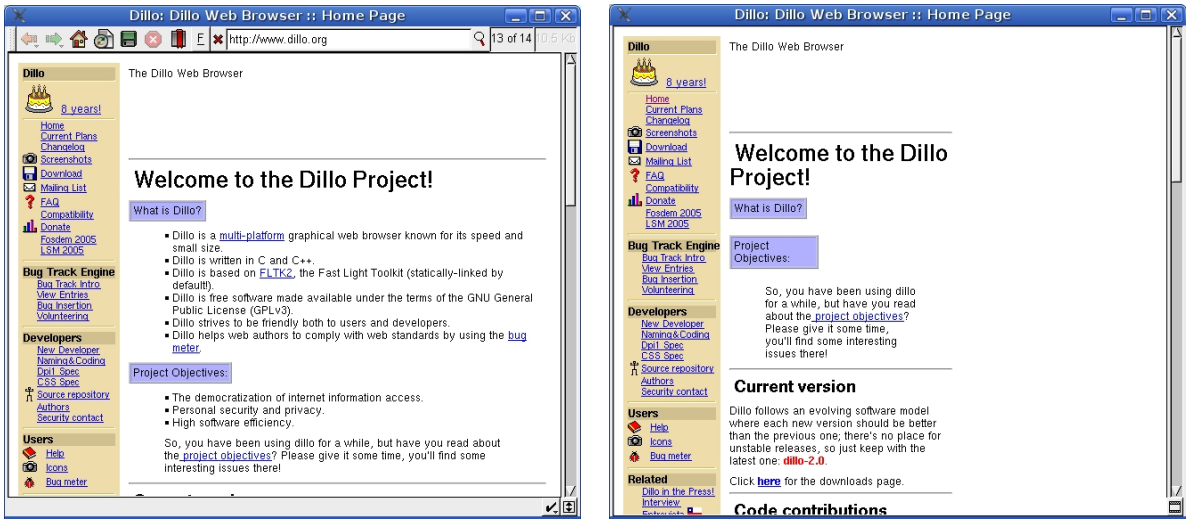
With Dillo, we followed the same template as with `sed`; however, with Dillo we had a much larger search space: at minimum, $2^{51}-2$ (see Table 2). We used a population size of 30 over 20 generations, two-point crossover, 90% tournament probability, and an elitism size of 8.

As with `sed`, there was not much reason to run the GA for many generations as we had a fitness function with a maximum score of 7. The fitness function only checked for basic viability: rendering a simple HTML file from disk, rendering an Internet site that contains only HTML, and rendering a site with HTML and JavaScript. No attempt was made to verify that proper output was produced; instead, screenshots were captured for later analysis.

As Table 2 shows, while some version crosses produced many unstable variants, a large number were stable. These stable ones had a wide range of morphologies. A few appeared to be perfectly functional; others would only bring up an empty window. With some, it was possible to view the loaded HTML source even though nothing was rendered on the screen. Some could only render part of a page. Some started with toolbars hidden, while others started with toolbars visible. One particularly interesting variant is shown in Figure 3. Here, a cross between Dillo 0.8.4 and 0.8.1 produced a browser that would successfully load and render pages, except that unordered lists were omitted and the page width specification was ignored.

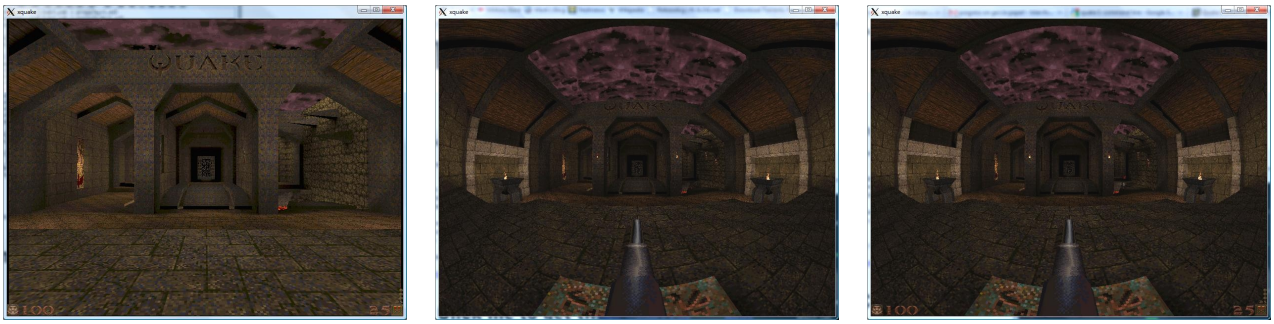
4.3 Quake

Dillo showed us that object-level recombination would also work on larger, more modern desktop applications. These experiments also indicated to us that we were unlikely to get many useful variants by combining older and newer versions of a program; instead, we'd more likely find versions with



(a) (b)

Figure 3: (a) Dillo 0.8.5 showing the Dillo home page, and (b) a Dillo variant generated from versions 0.8.4 and 0.8.1 rendering the same page. Note how (b) omits HTML unordered lists and ignores the width parameter of an HTML table when rendering the page.



(a) (b) (c)

Figure 4: Screenshots of Quake variants. (a) MakaQuake with a heads-up display (HUD) but no models (note the missing torches and fires), (b) Fisheye Quake variant with working models, and (c) an automatically recombined Quake with fisheye, HUD functionality, and working models.

Version	Files	Symbols	Symbols
TyrQuake	92	1807	375
FishEyeQuake	89	2104	12
MakaQu	89	2160	450
SDLQuake	92	1962	168

Table 3: A comparison of the versions of Quake used. "Symbols" represents the total number of symbols of all the object files for a particular version. "Symbols" is the total number of symbols that are different compared to the *original* Quake source.

Version Pair	Files	Unique	Unstable	Symbols
Tyr × SDL	84	727	204	459
Tyr × FishEye	88	738	138	387
Tyr × MakaQu	88	679	292	671
SDL × FishEye	85	676	149	180
SDL × MakaQu	85	476	64	428
FishEye × MQ	89	728	93	462

Table 4: Results for the tested Quake version pairs. 810 individuals were generated for each pair. "Symbols" indicates the total number which are not common between the two versions. This is used to provide a crude measure of how closely related the versions might be.

interesting bugs. This insight was our motivation to look to programs with multiple, divergent versions already existing. Thus did we start looking into Quake variants.

After experimenting informally with many versions, we decided to do our experiments with four variants:

- FishEyeQuake, a Quake version that renders the world as viewed through a fisheye lens, `strlen.com/gfxengine/fisheyequake/index.html`
- SDLQuake 1.09, a port of Quake to the SDL, a cross-platform multimedia library, `www.libsdl.org/projects/quake/`
- TyrQuake 0.38, a version with many bugfixes and code cleanups, and `disenchant.net/engine.html`
- MakaQu 0.1[3], a version with a revamped menu system, a "heads-up display" status bar, improved save game system, and major OpenGL rendering changes. `quakedev.dcemulation.org/fragger/downloads.htm`

To evaluate fitness, we compared the standard output from a demo run of 'vanilla' Quake, scoring as we did with `sed`. We used single-point crossover, 90% tournament probability, an elitism value of 10 with a population size of 50 for 20 generations. Again, we did not seek to find optimal individuals but interesting variants.

The versions that were the least divergent, SDLQuake and FishEyeQuake, were able to produce a large number of working variants. What was interesting was how quickly the

[3]Official build is supported on Windows only; modification is required to build on Linux

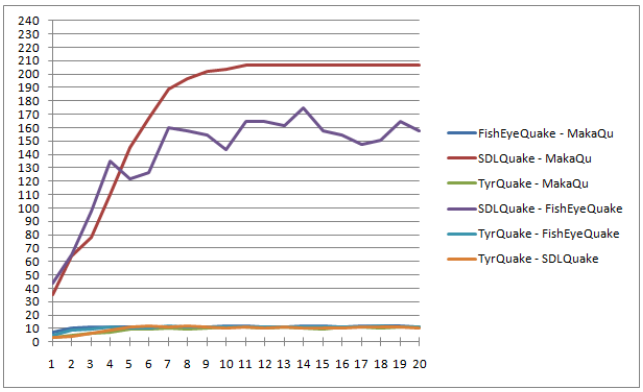


Figure 5: Average fitness achieved at each generation for the recombined versions of Quake.

population converged on working solutions. The test started with a number of individuals at discrete fitness levels: 2, 55, and 175. Within seven generations most individuals had achieved a fitness of 175 out of a possible maximum of 240 (see Figure 5). This population had not converged on a few individuals, however, as over the run ObjRecombGA found 468 unique individuals with a 175 fitness value, many of which derived 55-60% of their genome from one parent. This finding was not all that surprising because SDLquake and FishEyeQuake had the least divergent code bases: only 93% (85) of the object files were shared, but only 9% (180) of the symbols were different. The randomly initialized population contained a large number of unfit individuals; recombination of those individuals, however, quickly led to a large number of relatively functional variants.

SDLQuake and MakaQu also seemed to be very successful at recombination, producing the least number of unique variants but offering the highest average fitness scores. However, this result turned out be misleading. While the variants did load and run successfully, they exited prematurely in a clean fashion (leaving no clear indication of a fatal error). This resulted in what appeared to be correct output and lead to a high degree of fitness, when in reality the variants were not actually playable.

In terms of interesting functionality, however, we got some of the most interesting results from a cross of FishEyeQuake with MakaQu. As shown in Figure 4, FishEyeQuake can render objects in the world (models) properly and gives a fisheye view of the world. MakaQu on Linux, however, had severe glitches: it did not support models, so it could not even render the torches in the initial map properly. However, it did have a nice heads-up display. By combining the two we were able to get a variant that had a heads-up display, working models, and the 'fisheye' view. In other words, we were able to repair bugs and merge functionality by recombining programs at the object file level.

5. RELATED WORK

Many researchers have studied the problem of evolving programs. The bulk of the work in this area has followed the template of Koza's genetic programming (GP) [9], specifically in the representation of programs using S-expressions consisting of problem-specific functions and terminals. Work on grammatical evolution [11], however, has adapted the GP

representations to store programs encoded in arbitrary computer languages—programs are parsed into S-expressions using a supplied grammar. While grammatical evolution has been successfully applied to C programs [10], these programs have been either evolved from scratch or from handcrafted, relatively small progenitors.

Over the past few years, however, a few researchers have started looking into applying evolutionary techniques to commodity programs. One particular work of note is Arcuri's JAFF [3]. JAFF can repair faults automatically in Java programs by identifying parts that fail unit tests and evolving limited software patches. While JAFF can currently run only on programs expressed in a subset of Java, the approach may be extendable to work on arbitrary Java programs.

Last year's work by Forrest et al. [7] is the only work we know where genetic programming has been used to directly evolve the code of commodity programs. Given precise positive and negative test cases describing desired functional characteristics, they were able to successfully evolve security and bug fix patches for 20,000+ line C programs.

A number of artificial life systems have been developed that evolve machine code representations, most notably Core War [6], Tierra [13], and Avida [2]. Unlike ObjRecombGA, however, they all work with programs encoded in bytecodes designed to facilitate evolutionary search rather than the object files produced by standard compilers.

6. CONCLUSION

We have shown that it is feasible to recombine related programs at the object-file level using a genetic algorithm framework and careful resolution of symbol dependencies between object files. Many program binaries produced by this method do not function properly; however, many do. Our success to date gives us significant hope that these techniques could be used to aid with debugging, improve security, and ultimately enable user-directed software specialization. Much as animals are bred to produce desirable individuals, it should be possible to the same with our desktop programs. Further testing, however, is needed to determine the full applicability of these techniques.

7. ACKNOWLEDGMENTS

This work was originally inspired by discussions at a June 2005 workshop at the Santa Fe Institute entitled "The Road to Software Evolvability." Somayaji in particular would like to thank David Ackley and his graduate advisor, Stephanie Forrest, for their contributions at this meeting and in discussions before and since.

This work was supported by the Discovery Grant program of the Natural Sciences and Engineering Research Council of Canada (NSERC).

8. REFERENCES

[1] D. Ackley. Real artificial life: Where we may be. In *Artificial Life VII: Proceedings of the Seventh International Conference on Artificial Life)*. MIT Press, 2000.

[2] C. Adami and C. Brown. Evolutionary learning in the 2d artificial life system 'avida'. In R. Brooks and P. Maes, editors, *Artificial Life IV*, pages 377–381. MIT Press, 1994.

[3] A. Arcuri. Evolutionary repair of faulty software. Technical Report CSR-09-02, University of Birmingham, 2009.

[4] V. Authors. *Seder's Script Archive*, 2008. http://sed.sourceforge.net/grabbag/scripts/.

[5] R. Dawkins. *The Blind Watchmaker*. Norton, 1986.

[6] A. Dewdney. In the game called core war hostile programs engage in a battle of bits. *Scientific American*, May 1984.

[7] S. Forrest, T. Nguyen, W. Weimer, and C. L. Goues. A genetic programming approach to automated software repair. In *GECCO '09*, 2009.

[8] F. S. Foundation. GNU `sed`. http://www.gnu.org/software/sed/.

[9] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

[10] M. O'Neill and C. Ryan. Evolving multi-line compilable c programs. In *1999 European Conference on Genetic Programming (EuroGP'99)*, 1999.

[11] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[12] T. D. Project. Dillo: the fast and light browser. http://www.dillo.org/.

[13] T. Ray. An approach to the synthesis of life. In C. Langton, C. Taylor, J. Farmer, and S. Rasmussen, editors, *Artificial Life II*, pages 371–408. Addison-Wesley, 1991.

[14] I. Software. Quake. http://www.idsoftware.com/games/quake/quake/, 1996.