Lancaster University

Experiments in Genetic Divergence for Emergent Systems

by **Christopher McGowan, Alexander Wild and Barry Porter**

School of Computing & Communications
Lancaster University

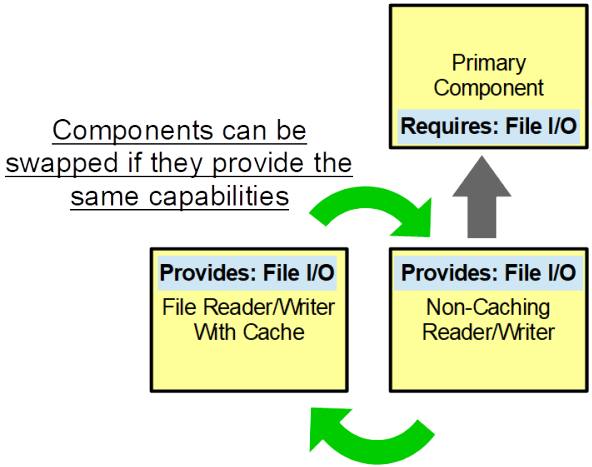
School of Computing and Communications

Motivation

As software becomes more complex, and the user's hardware and task requirements become more diverse, software must adapt itself locally, rather than depend on its designer

Emergent Software seeks to produce solutions which can adapt themselves to their environment

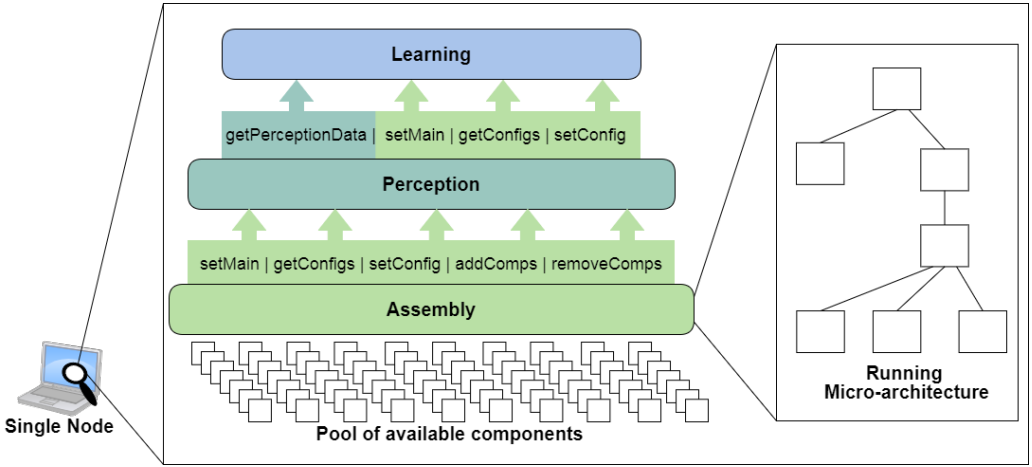
The DANA Language



DANA's emergent architecture

DANA components are assembled into tree-like structures, with each node requiring the creation of a set of subordinate nodes, until terminal nodes are reached which require nothing

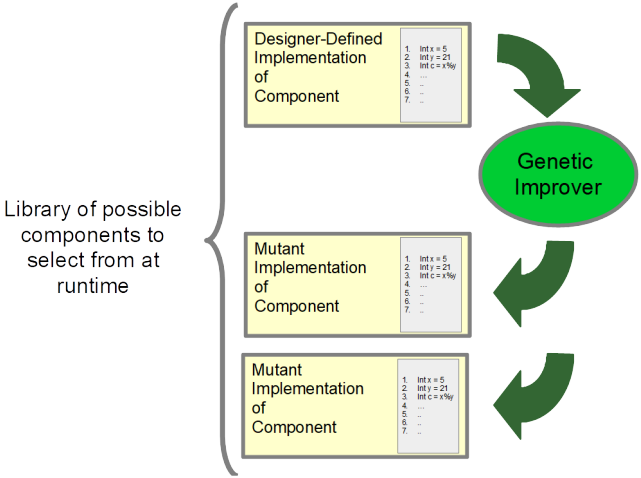
DANA's learning capabilities for self-assembly



Emergent software built around Genetic Improvement

- Genetic Improvement of components is required to fully deliver emergent software, as it allows full self-adaptation of the program
- Designer builds templates, which perform a function in a generic way
- Code monitors its own performance post-deployment, on user environment and user workload
- New components are created to suit these new workloads, using genetic improvement, with the designer-created templates as starting points
- Emergent Architecture learns to use these components based on its empirical assessment of their performance

Genetic Library



Advantages of genetic improvement in DANA

Component based structure means a set of mutations can be performed on a specific part of the system

- Shorter code blocks reduces range of possible mutations, allowing better search performance
- Allows easy evaluation of performance, as a known part of the system is being targeted without altering the remaining
- Allows serial improvement of the system, one component at a time
- Allows online swapping in of improved components from a background improvement process without loss of uptime

Contributions to the field

- First investigation into combination of the concepts of genetic improvement of source code and emergent software
- Demonstrate ability of an algorithm to produce a usable variant to an existing implementation
- Investigation into using full code synthesis capabilities for Genetic Improvement
- Implement a Genetic Improvement mechanism which alternates between a "synthesis" and an "improvement" phase, defined by different fitness functions

The Plastic Surgery Hypothesis

The plastic surgery hypothesis (Barr et al, 2014) suggests that mutations for genetic improvement can take the form of "grafts" from other parts of the source code

This allows complex code to be inserted as mutations, without allowing full range of code synthesis (only copying existing complex structures). This limit reduces search space therefore improving search speed

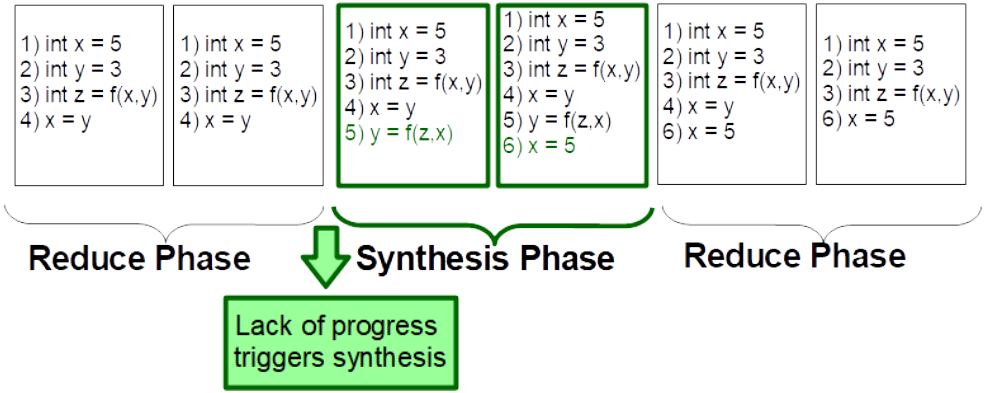
Going Beyond the Plastic Surgery Hypothesis

Smaller code elements, with fewer lines to mutate, allows for greater range of mutations. We investigate full code synthesis, in opposition to the often-used mutate-by-plastic-surgery approach

Issues with code length as part of the fitness function

- Shorter code is classically preferred, with redundant lines removed, and the fitness function written to encode this preference
- If multiple lines are required to be added to produce any improvement in fitness, the algorithm must avoid penalising them or they will be rejected
- Logic synthesis must allow the code to grow, to allow neutral mutations to persist

Synthesis and Reduction phases



Specifics: Our Task

We focus on the problem of optimising a component which provides a hashing function, which provided functionality for a cache handling component in a web server

- Evaluated based on uniformity of bucket filling
- Requires maximisation of entropy for the input set
- Performance depends on the nature of the input in terms of the patterns seen, so good candidate for user-side optimisation of component

Implementation: Synthesis vs Reduce Phase

- Evaluate populations as in classic genetic improvement, with penalty for code length
- Once 5 generations have occurred with no fitness gain, switch to "Synthesis" phase
- Code length no longer taken into account for fitness, allowing code to expand
- Synthesis phase continues for 6 generations, then ends
- Code now penalised for length, begins to reduce again, while preserving the valuable new mutations

Implementation: Mutations

A range of mutations were implemented, allowing for full logic synthesis. These were:

- Declare Variable from available type, including assignment expression, which includes function calls
- Assign to Variable using expression
- Add flow-control operation
- Replace token with equivalent, including allowing expanding a single variable use in expression into a nested set of function calls
- Add return statement
- Delete line

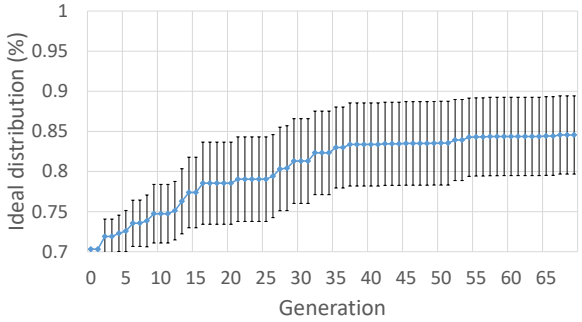
Implementation: Selection and Crossover

Other aspects of our implementation to note are:

- Standard roulette selection was used for selecting parents for genetic reproduction
- Elitism with 2 population members
- Population size of 14
- Crossover by sequentially selecting lines from either parent, preserving entire blocks of code if control statements are chosen

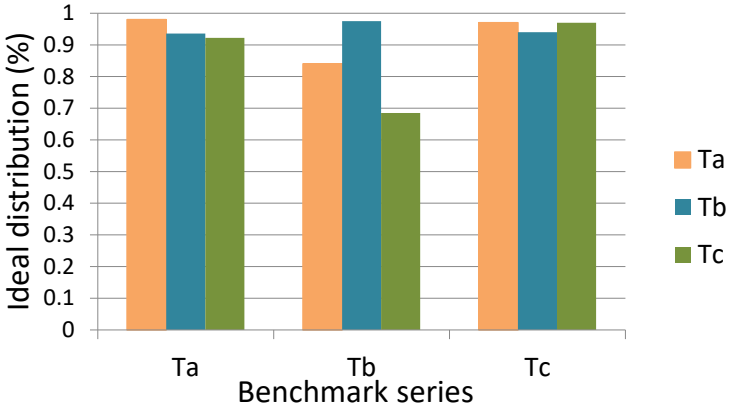
Results

Aim was to produce a function which distributed incoming strings in a uniform way between hash-buckets. 100% indicates perfectly equal distribution between all buckets



Results on different benchmarks

Different input sets were compared, to evaluate how much specialisation could improve performance for each user's data



Future Work

We hope to continue to work on the problem of combining genetic improvement with emergent software. Lines of investigation we are considering include:

- Automating the process of input-output example collection and subsequent component generation for fully hands-free improvement
- Learnt probabilities per mutator, to produce higher fitness mutants
- Learnt biases for tokens used by mutators, based on position within program and surrounding lines of code

Conclusion

In conclusion, our work demonstrated an approach towards the use of Genetic Improvement to bring about fully emergent software, which is able to adapt itself locally to the user's demands and requirements, without need for designer involvement

Code available for results replication
(`http://research.projectdana.com/gi2018mcgowan`**)**