Exploring Automated Software Composition with Genetic Programming

Erik M. Fredericks and Betty H. C. Cheng
Michigan State University
East Lansing, Michigan 48824-1226, USA
{freder99, chengb}@cse.msu.edu

ABSTRACT

Much research has been performed in investigating the numerous dimensions of software composition. Challenges include creating a composition-based design process, designing software for reuse, investigating various strategies for composition, and automating the composition process. Depending on the complexity of the relevant components, numerous composition strategies may exist, each of which may have several options and variations for aggregate steps in realizing these strategies. This paper presents an evolutionary computation-based framework for automatically searching for and realizing an optimal composition strategy for composing a given target module into an existing software system.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Algorithms

Keywords

Search-based Software Engineering, Genetic Programming, Software Composition

1. INTRODUCTION

Incorporating new capabilities into legacy software systems can be a non-trivial task. Environmental constraints and new/updated requirements must be considered when introducing modules to update system functionality. Software composition is an approach for building and maintaining large software systems by integrating existing software modules. Given the numerous factors that must be considered when composing *target* modules into an existing software system (i.e. *source* module), automating some or all of this process is an attractive option. This paper overviews an approach that harnesses evolutionary computation to automate the composition process.

Software composition is an approach for creating a software system by incorporating existing objects or modules to form a larger composite system [6]. Research into automated software composition has yielded promising results by applying *superimposition* [1], a process that merges the substructures within software modules. Nevertheless, as software systems grow in complexity it can become necessary for search-based heuristics, specifically evolutionary computation [2, 4, 5], to take part in the code generation process in order to efficiently and intelligently expedite code reuse.

This paper presents the SAGE (Software Adaptation Genetically Engineered) project, an approach for facilitating software composition with evolutionary computation. SAGE searches for possible combinations of steps necessary to instantiate and perform a successful and efficient composition strategy given *source* and *target* modules. By incorporating evolutionary computation at the code-level, SAGE is able to automatically discover possible composition solutions that may not have been previously considered.

SAGE applies genetic programming (GP) [2, 5] to efficiently search for optimal realizations of composition strategies. GP is an evolutionary algorithm that automatically generates programs for a defined task and is traditionally represented by an abstract syntax tree or a stack [7]. In particular, SAGE generates a tree that specifies the ordered operations of a specific composition approach. These operations have been defined based upon composition strategies as identified by a requirements engineer. Furthermore, fitness criteria drive the search process to create programs that ensure *structural* compatibility by analyzing method signatures and pre- and post-conditions.

This extended abstract is structured as follows. Section 2 discusses the SAGE approach in further detail, and Section 3 summarizes our current results and proposes future work.

2. SAGE APPROACH

This section presents current research in the development of SAGE, a GP approach for automatically composing software. SAGE is built upon OpenBEAGLE-Puppy [1], a derivative of the OpenBEAGLE [3] framework for evolutionary computation. Puppy is a lightweight and extensible tree-based framework specifically for GP.

The SAGE genome comprises a set of composition operators that supersede the standard arithmetic GP operators. These operators reflect necessary operations in performing a composition that depend on the target language to be composed, where we currently target the C programming language. Table 1 provides a summary of the composition

[1]Available at: `http://code.google.com/p/beagle/wiki/Puppy`

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.
ACM 978-1-4503-1964-5/13/07.

operators that have been identified thus far as necessary to perform a composition in C. Each is a high-level representation of an operation to be performed and may have various parameters, such as values or code statements.

Table 1: SAGE Composition Operators.

Operator	Description
Wrapper	Encapsulate composed module
Function pointer	Provide reusable invocation interface
Code Injection	Directly insert code into source module
Transform	Transform data into specified format

While the identified operators in Table 1 provide the basis for the genome in the evolutionary process, it is also necessary to provide fitness criteria to evaluate the generated composed software. SAGE ensures structural compatibility through formal analysis of pre- and post-conditions. Given a *source* module and a *target* module to be composed, SAGE will process the pre- and post-conditions of both and define the necessary composition steps to ensure the *target* module can be safely invoked and that the post-conditions of the *source* module are satisfied. Furthermore, SAGE also ensures that invocation requirements are satisfied by analysis of method signatures. A role is assigned for each parameter in both *source* and *target* modules to guarantee that corresponding parameters are used to exchange information between modules.

Consider a program that requires a large amount of records to be sorted in an efficient manner, for instance, a financial institution may need to sort daily transactions or a university may need to sort student records. In a typical composition, a software engineer may choose from several existing sorting algorithms and must ensure that (1) the array to be sorted is properly delivered to the selected *target* sorting module and (2) that the output of the *target* module returns a sorted array of numbers to the *source* module. Analysis of method signatures and proper definition of parameter roles satisfies (1), and satisfaction of post-conditions handles (2).

Figure 1 depicts a simplified sample genome using the composition operators identified in Table 1. This genome was tasked with composing a bubble sort operation into an existing program. SAGE explores numerous composition solutions, and eventually the solution with the best fitness value (1) injects a (2) wrapper at line 144 within the *source* module. This wrapper will then directly invoke the `bubble_sort` method with an integer pointer parameter. Upon completion of the `bubble_sort` invocation, a properly sorted array is then returned to its *source* module.

3. CONCLUSION

This paper introduced SAGE, a GP-based approach for automatically composing software at the code-level. SAGE comprises of a set of composition operators that define the possible steps that a composition strategy may use, including transformation, wrapping, and code injection. These operators are then incorporated into a GP framework in order to efficiently explore the space of structurally-compatible compositions. At present, SAGE has been configured specifically for the C programming language.

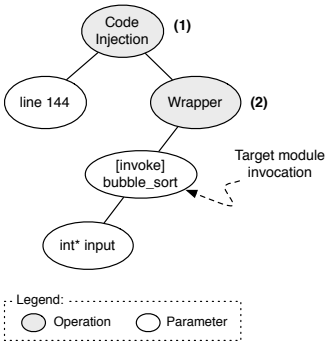

Figure 1: Sample genome for composing a bubble sort module.

Our initial explorations into automated software composition using evolutionary computation have been promising. As such, we are expanding the genome of SAGE to tackle increasingly more complex applications as we include behavioral, as well as structural, compatibility in determining genome fitness. Finally, we also plan to extend SAGE to support other high-level languages, as they may introduce new composition operations.

4. ACKNOWLEDGMENTS

This work has been supported in part by funding from Air Force Research Laboratory (AFRL) and Defense Advanced Research Project Agency (DARPA). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the AFRL, DARPA, or other research sponsors.

5. REFERENCES

[1] S. Apel, C. Kastner, and C. Lengauer. Featurehouse: Language-independent, automated software composition. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conf. on*, pages 221–231, Vancouver, BC, Canada, 2009. IEEE.

[2] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of an International Conf. on Genetic Algorithms and the Applications*, pages 183–187, Pittsburgh, PA, USA, 1985.

[3] C. Gagné and M. Parizeau. Open beagle: A new versatile c++ framework for evolutionary computation. In *Proceedings of GECCO*, New York, NY, USA, 2002.

[4] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, MA, USA, 1992.

[5] J. R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, 1994.

[6] K.-K. Lau and T. Rana. A taxonomy of software composition mechanisms. *Proc. 36th EUROMICRO SEAA*, pages 102–110, 2010.

[7] T. Perkis. Stack-based genetic programming. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conf. on*, pages 148–153. IEEE, 1994.