A New Approach to Data Mining for Software Design*

Walid Taha, Scott Crosby, Kedar Swadi
Department of Computer Science, Rice University,
6100 S. Main St, Houston, TX 77025 USA.
{taha,scrosby,kswadi}@rice.edu

Abstract

We propose Exact Software Design Mining (ESDM) as a new approach to managing large software systems. ESDM is concerned with automatically extracting a program generator capable of exactly reconstructing the full code base. To illustrate the potential effectiveness of this approach, we propose, implement, and test a basic algorithm extracting such generators. Experimental results on various benchmarks (including the Linux kernel) are encouraging.
Keywords: Software engineering, software design, program generation.

1 Introduction

Over the last decade, data-mining has emerged as an important technology with significant applications in database systems and bioinformatics problems. Almost in parallel, program generation enjoyed a renewed interest as a technology with significant software engineering applications. But the exact role for program generation in software engineering has remained a somewhat elusive goal. We propose a view motivated by Kolmogorov's notion of string complexity [9]: Generators can capture the essential complexity of the programs they produce. At the same time, abstractions used in the generator do not force any runtime overhead on the final product. If such generators are both compact in size and readable, they provide an effective way to capture the *design* of software. They also capture an *exact* design, in that it contains all the information needed to reconstruct the full code base.

Our thesis is that data mining techniques can be used to extract such generators. This paper argues that systems that support this thesis, or Exact Software Design Mining (ESDM) systems, are both feasible and challenging. The paper is organized as follows: Section 2 provides a broader introduction to program generation and discusses techniques that have similarities in goals to ESDM. Section 3 presents an example of an ESDM system that uses common-subexpression elimination (CSE) to extract a generator from a software base. An implementation of this algorithm for the full C language is described. Experimental results obtained with this implementation are presented in Section 4. Section 5 outlines a broad range of extensions that can be made to this basic ESDM method, and that can be expected to yield improvement to the design extraction process. Section 6 concludes.

2 Background and Related Work

Program generation is an increasingly important technique for software development — important enough that an annual ACM conference has recently been established with this technique as its focus [2, 5]. But a basic question remains open: why does program generation help software development? Compiler [7] and partial evaluation [8] researchers generally argue that the point of program generation is improved performance. While these observations explain many instances where program generators are useful, they are not universal, and do not necessarily explain important cases when the generation of high-level programs are useful. In particular, partial evaluation produces greatly variable performance improvements, and it is not easy to predict ahead of time how much performance will be gained.

We postulate that the utility of high-level program generation techniques lies in that they allow the programmer to write programs using more powerful abstraction mechanisms than was possible before, and

*Supported by NSF ITR-0113569 "Putting Multi-stage Annotations to Work" and Texas ATP 003604-0032-2003 "Advanced Languages Techniques for Device Drivers."

without having to pay a runtime performance overhead for these abstractions. This view explains both why program generation can be used when there is no runtime performance benefit, and why techniques such as partial evaluation are very useful despite the fact that a programmer can always write the code they generate by hand. There are several reasons why software written without generation is often impoverished in terms of abstractions:

- The language lacks the abstraction mechanisms. For example, a first-order language cannot express call-backs, and so if we want to implement a sort function using different order functions, we would have to write multiple copies of the sort function.

- The language being used may support an abstraction mechanism, but using the abstraction mechanism would entail a runtime performance overhead. For example, in a statically typed language, functions for traversing different types usually have to be expressed explicitly, because there is no facility for taking the definition of a type and generating such functions.

Almost forty years ago, Kolmogorov suggested that the information-theoretic complexity of a *string* can be defined as the *length* of the smallest generator that would produce this string when executed [9]. In general, Kolmogorov complexity is not computable, but computable approximations have been developed and studied, often for the purpose of string compression [11]. Another application of Kolmogorov complexity to a domain that is closely related to our work is in software plagiarism detection [4].

There are similarities between plagiarism detection and ESDM, but there are also important differences. The differences can be illustrated with a concrete example. Consider a class of students that are given a programming assignment. The goal of plagiarism detection is to identify which subsets of the solutions are similar. Ideally, subsets that are identified as similar are ones where plagiarism has occurred. Alternatively, we can view the output of plagiarism detection as a probability of plagiarism between any two assignments. In fact, this is the form of the output of such systems. In contrast, the ultimate goal of software design discovery is to first recognize that all solutions are similar, and then to explain precisely how each implementation can be mechanically reconstructed starting from a core design and ending with each different implementation. At a more technical level, the notion of plagiarism must generally limit the notion of similarity between programs to trivial textual edits, otherwise, all correct solutions to the homework problem would be considered plagiarized [14]. For design discovery, the only limit is computability. In fact, if programmer intervention is allowed, even this limit can be relaxed.

But the exact *form* of the smallest generator for a string has only recently been a subject of interest, and much of this work has taken place in the context of data-mining research. A particularly promising approach in the field of data-mining has been the SEQUITUR system. Given a string, SEQUITUR computes a context free grammar that generates exactly this string [13]. The grammar is computed in linear time. From the point of view of extracting artifacts that capture the design of a software system exactly, the fact that grammars can be readable objects suggests that this approach can be used to extract from a large code base a more compact program generator that can be read, understood, and maintained instead of the full codebase. From the complexity theoretic point of view, SEQUITUR has some limitations: it can only approximate Kolmogorov complexity up to a factor of $O(n^{1/3})$. Note, however, that this is a limitation of SEQUITUR's approach: other techniques based on context free grammars have a lower bound of $O(log(n/g^*))$, where g^* is the size of the smallest grammar [3].

3 Extracting Generators

By and large, data-mining and compression algorithms work at the level of tokens or characters and not abstract syntax trees (ASTs). This has two important implications: First, token-level algorithms are fundamentally prone to accidental similarities at the level of tokens that is meaningless at the grammatical level. Second, and more importantly, such algorithms have to work hard to discover structure that is obvious at the level of the AST.

To address the problem described above, we propose the use of a simpler but more structured technique of common-subexpression elimination (CSE) [6, 1] for dealing with data that has well-defined structure. In particular, a powerful feature of CSE on ASTs is that it gives optimal compression on duplicates: Size is only increased by a constant factor, as exactly one extra node is needed per duplicate. This is not the case for state of the art compression algorithms such as Compress, Gzip-9, Bzip-2, PPM-9, and Token Compress [4].

As a practical approach to assessing the potential of ESDM, we studied and implemented a basic method for extracting generators based on common-

subexpression elimination (CSE) [6, 1]. Compilers sometimes use CSE to improve the runtime performance of programs by avoiding duplication of work. Because CSE does not always improve runtime performance (because inlining an expression can sometimes improve it), compilers must apply CSE selectively. For ESDM, we will use full CSE. In particular, the generators we extract will always inline all common expressions to produce exactly the original program. All of this happens before the program is compiled and executed. So, ESDM has no impact on compilation times or the runtime behavior of programs, because compilation starts from the same sources as before. So, whereas complex heuristics have to be used by compilers to determine where CSE should be used, full CSE can be used for ESDM. CSE can be implemented efficiently by hash-consing [6] followed by inlining of trivial terms, that would run in almost-linear time ($O(n \cdot log(n))$). That CSE works by searching for redundancy and eliminating it in a form that is still readable suggests that it is a natural starting point for investigating ESDM.

3.1 Implementing CSE

We implement CSE by performing hash-consing followed by inlining of trivial terms. Hash-consing [6] is a standard technique for compactly storing a tree as a directed-acyclic graph (DAG) with maximal sharing. To apply it to an AST, we begin with the leaves, name them, and then process the rest of the nodes from the bottom up. For each node that we see, if a node of the same type and the same branches has already been named, then this node is replaced by its name, and we move up to the next level. Named terms generated in this manner can be viewed as a grammar. For a given expression such as `g (sqrt (7234) * sqrt (7234))`, the resulting grammar would be

$$
\begin{array}{l}
A \rightarrow \texttt{g} \\
B \rightarrow \texttt{sqrt} \\
C \rightarrow \texttt{7234} \\
D \rightarrow B(C) \\
E \rightarrow D{*}D \\
F \rightarrow A(E)
\end{array}
$$

This grammar represents a DAG with maximal sharing. But trivial rules that produce only one terminal (such as A, B, and C) or rules that are only used once (such as E) inflate the size of the grammar unnecessarily, without improving sharing. All nonterminals satisfying either of these two conditions can be inlined without increasing the overall size of the grammar, and possibly reducing it. The result of this step is also guaranteed to have at most as many symbols as the original program. For the example above, the result is

$$D \rightarrow \texttt{sqrt}(\texttt{7234}), \quad F \rightarrow \texttt{g}(D{*}D)$$

3.2 Grammars as Program Generators

Because the resulting grammars are DAGs, they can be sorted so that each production only depends on productions that have been seen before, and each production can be viewed as a program that produces a string. For example, the grammar can be used to build the following macro program:

```
D = sqrt (7234); F = g (D * D)
```

When executed, this program *generates* precisely the term that we started with: `g (sqrt (7234) * sqrt (7234))`.

3.3 Independence of Variable Names

Syntactic and semantic equivalence can be unrelated. Programs sometimes have textually identical but semantically unrelated fragments. For example, occurrences of terms such as `a+b` are tokenwise identical, but `a` and `b` might happen to occur in different contexts that provide different bindings to each of these variables. On the other hand, textually different fragments can be semantically identical. For example, the two function definitions `f(x)=x+x` and `f(y)=y+y` are identical except for the variable names. To address both these issues, we apply a standard transformation to ASTs, which eliminates variable names from the source programs and replaces them by the de Bruijn index (or, static distance coordinates) of the parameter. The result of this transformation would convert both of the function definitions above into `f(_) = #0 + #0`, where the notation `#0` denotes the de Bruijn index corresponding to the variables `x` and `y`. If the contexts provide different bindings to variable names in textually identical fragments, the result of de Bruijn conversion would be different (with fragments having different de Bruijn indices), and the accidental matching would be avoided.

When extracting the generator from the grammar, de Bruijn indices are replaced by names and lookups in generation-time environments.

3.4 Implementation

Assessing the practical utility of ESDM in general, and the approach proposed above in particular, requires empirical analysis of large, realistic software systems. This is particularly important because, to

Name	Size	Direct	De Bruijn	Time	Bytes	≈S
xpm	79605	1.2%	1.2%	6.59	657K	27%
gfig	121138	20.7%	19.6%	8.17	934K	26%
UI	116023	4.1%	4.0%	13.10	915K	26%
vmlinux	1957357	45.7%	43.3%	310.31	12,572K	34%

Figure 1: Experimental results

our knowledge, little work has been previously done on gathering and understanding patterns that naturally occur in software systems. To this end, we have designed and implemented an instance of the above generator extraction algorithm tailored for the C programming language using the parser that comes with the CIL program transformation system [12].

CSE is implemented as a direct extension to the type of the AST produced by CIL's [12] parser for C. Conversion of variable names to de Bruijn indices is only done for function parameters, local variables in functions, variables in block scope, and in type declarations. As an example, consider the following function:

```
int f (int a) {
  int b;
  b = 10;
  {
    int c = 20;
    c = c + b + a;
  }
  return 0;
}
```

At the assignment statement `c = c + b + a;`, the closest binding variable, `c` has the de Bruijn index 0 , variable `b` has the de Bruijn index 1, and the function argument has the de Bruijn index 2. A simplified AST for the substituted statement looks like: `ASSIGN (#0, ADD (ADD (#0, #1), #2))`. Global variables are left untouched, and no de Bruijn indices are generated for their occurrences in the programs.

As an example of type declarations, consider the declaration

```
typedef struct { int val; } s1;
typedef int * ip;

typedef struct ablist {
   ip a;
   s1 b;
   struct ablist * c;
} s2;
```

In the structure `ablist`, there are occurrences to three named types, namely, `s1`, `ip`, and a recursive occurrence of a pointer to itself (`ablist`). In a manner similar to values, we also replace occurrences of such types with de Bruijn indices for types. As a result, the type `s2` above is represented in a simplified AST as: `STRUCT{a: #1, b: #2, c: PTR #0 }`.

4 Experimental Results

Figure 1 summarizes experimental results with using the implementation described above on some non-trivial input examples. The first column contains the name of the program analyzed. The first three are components of the GIMP [10] system. The last program is a merged version of the Linux kernel [12]. **Size** counts the number of nodes in the original AST before CSE. **Direct** shows the relative reduction in size after CSE. **De Bruijn** shows the relative reduction in size when the AST is converted into de Bruijn form and then hash-consed. **Time** gives the amount of time in seconds required to execute [1] our algorithm (with both the hash-consing and de Bruijn indexing). **Bytes** reports the size of the source file in bytes. Time is only included to confirm that the implementation does run in time linear to the size of the input. The last column, titled ≈**S**, reports the relative reduction in number of bytes in the standard SEQUITUR [13] output (`sequitor -p`). This output is itself a grammar similar to what we generate, but is produced without knowledge of the actual grammar of the C language, and is printed into a file.

The percentages for CSE indicate that compression rates can vary significantly depending on the program, but that they can also reach over 40% for large, real, programs such as the Linux kernel. Interestingly, translating programs into de Bruijn form did not help compression, but actually consistently caused a small degree of deterioration. Nevertheless, we expect conversion into de Bruijn form to enable easier processing and implementation of more

[1] Timings are measured on an AMD XP-2500+ machine with 1.5GB RAM running Linux version 2.6.7 under the OCaml interpreter version 3.08.1

equalities in the future. Examples include change in name combined with change in order of arguments. Further experimental results (not shown in the table above) indicate that there is large variability between the amount of compression when we consider only expressions, statements, definitions, or type declarations.

The last two columns are included only for reference, and comparisons between **Direct** and $\approx$**S** should be made with care. In particular, **Direct** is a ratio based on node counts, while $\approx$**S** is based on textual representations. Thus, the latter is only a rough estimate of ratio on node counts for SEQUITUR's algorithm. In particular, the numbers for $\approx$**S** can be increased arbitrarily by adding spaces to the original program. Thus, we expect that it is likely to be an over estimate of the performance of SEQUITUR on these inputs. It is also likely that a significant part of SEQUITUR's compression on these programs comes from compressing identifier names, which we do not consider desirable from the point of view of the readability of the extracted generator. What should be taken from the data in the table is that whereas the variation in the compression ratios for SEQUITUR on these examples is less than 10%, variation for CSE is about 45%. This suggests that they are rather independent measures of information content. The last case is most interesting, because it shows that the exact ratio for CSE is higher than what we expect as an upper bound for SEQUITUR's performance on the same benchmark. It suggests that our basic approach can in fact be superior to SEQUITUR for the particular domain of ESDM.

5 Directions for Future Work

There are many ways in which the basic algorithm described above can be extended, and which can lead to improvements in both the size and the readability of the result. We will classify these different approaches on the basis of the notion of equivalence that the user considers acceptable in the code base.

Syntactic equivalence: This means that we require that the extracted generator reproduces the source code base syntactically. The primary challenge here is to deal with inconsequential differences in the name and order of parameters, function declarations, field names, and type declarations. Note that converting names to de Bruijn indices, while helpful for immediately identifying terms that differ only in names of bound variables, also introduces some difficulties as well. For example, a term containing a function such as `f(x)` can have different de Bruijn representations depending on where it occurs in the program, even if both occurrences refer to the same `f` and `x` parameters. More generally, effective and efficient techniques for identifying equivalent terms will need to be developed, as well as techniques and formalism for managing the reconstruction of the original programs from the compressed representations.

This basic framework can support arbitrary equivalences. We expect that compositional equivalences that can be mechanically derived from one another are likely to be the most useful in practice. In particular, compositionality is likely to help keep the generator itself comprehensible to humans. Examples of such equivalences are the instantiation of patterns such as the visitor pattern in object oriented languages, or the map/fold patterns in functional languages. At the same time, the more sophisticated these patterns are, the more challenging the recognition of these patterns becomes.

Semantic equivalence: This means that we require that the extracted generator reproduces a semantic equivalent of the source code base. Thus, we are allowed to replace parts of the code base with equivalent subprograms. For C, such equivalences can include using `if` in place of `?`, interchange independent statements, simplify boolean expressions, or replace statements such as `i=i+1;` with `i++;`.

Deciding on what is an acceptable notion of equivalence: We must emphasize that the syntactic approach has the practical advantage of requiring only an understanding of the syntax of the language and *not* its semantics. In particular, for many widely used languages such as C, agreeing on a formal semantics is hard even for experts. Furthermore, depending on the application or domain, one may be interested in very different notions of semantics for the same language. For example, for Graphical User Interface (GUI) applications, we may be interested only in the input-output behavior of programs. In contrast, for real-time applications, we may be interested in the exact amount of time and space that is needed to perform a certain computation. Generally, the more we observe about program execution, the more complex the semantics becomes, and the fewer equivalences we have.

User interaction and execution-based extensions: Even if semantic equivalence is used, it must be approached with care. For example, allowing the user to introduce user-defined equivalences can introduce bugs into the code base, if these equivalences are invalid. The user, however, can in gen-

eral be queried to guide the system during extraction, as long as none of the options the user is offered can produce erroneous results. The fidelity of the system will depend on the appropriateness of the equivalences.

6 Conclusions

After motivating the need for Exact Software Design Mining (ESDM) techniques, this paper puts forth and studies one such technique. In the technique studied, common sub-expression elimination (CSE) is used to extract a program generator that reproduces the original code base. The extracted program generator can sometimes be 40% smaller than the original program. Experiments with an implementation of this technique verify that such significant reduction can in fact be the case for large, real, programs such as the Linux kernel. The paper outlines several directions and goals for further development of ESDM. Achieving these goals would have to a significant impact on how programmers approach the task of understanding and managing a large code base.

Acknowledgments We would like to thank Hisham Al-Mubaid, Corky Cartwright, Anthony Castanares, Narayan Debnath, Emir Pašalić, and Abd Elhamid Taha for valuable comments on earlier drafts.

References

[1] F. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[2] Don Batory, Charles Consel, and Walid Taha, editors. *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[3] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, April Rasala, Amit Sahai, and Abhi Shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proceedings of the thiryfourth annual ACM symposium on Theory of computing*, pages 792–801. ACM Press, 2002.

[4] Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. Shared Information and Program Plagiarism Detection. *IEEE Transactions on Information Theory*, 50(7):1545–1551, 2004.

[5] Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors. *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[6] A. P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–6, 1958.

[7] Matteo Frigo. A Fast Fourier Transform compiler. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 169–180, 1999.

[8] Neil D. Jones, Peter Sestoft, and Harald Sondergraard. An experiment in partial evaluation: The generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

[9] A.N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems in Information Transmission*, 1(1):1–7, 1965.

[10] Olof S. Kylander and Karin Kylander. *GIMP - The Official Handbook*. Coriolis Value, Nov 1999.

[11] Ming Li and Paul Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Verlag, seocnd edition, 1997.

[12] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228. Springer-Verlag, 2002.

[13] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences. *Journal of Artificial Intelligence Research*, 1997.

[14] A. Parker and J. O. Hamblen. Computer algorithms for plagiarism detection. *IEEE Transactions on Education*, 32(2):94–99, 1989.