Using a Very High Level Language to Build Families of High Quality Reusable Components

Gary F. Pollice
CenterLine Software, Inc. and
U. Massachusetts, Lowell

1 Introduction

While most programs are designed to perform a specific task, they have a natural evolution over time; causing a single program to become a set of programs that perform related tasks. In 1976 David Parnas introduced the concept of a *family of programs* [11]. He suggests that any program should be considered a member of a family of programs, all of which perform related tasks. If one plans for change when a program is designed less effort may be required to produce future revisions.

Today there is an emphasis on building software components for reuse. Components may be designs, programs, functions, classes, or code segments. Major reuse projects are in progress at several institutions, for example, the Software Productivity Consortium and the Software Engineering Institute [1, 6]. Processes have been developed to introduce and support reuse programs. The disciplines of domain engineering and software synthesis emphasize creating families of components and developing tools to assist in constructing them. Component generation is a prominent part of most efforts.

Until recently, the literature has focused on the processes. Generators are assumed to be available; however, good descriptions of what a generator looks like and how to build one have been scarce. Some exceptions do provide insight to useful generation techniques [13, 4, 5, 2].

The work described in this paper is a continuation of work done by the author [12] in which a framework for building component generators is presented. The goals were to be able to produce members of a family of components, each as good as hand coded components, and to determine if the techniques and architecture would generalize and scale up. The example used is a family of scanners, collectively known as **scangen**, for C and C++.

Metaprogramming, that is writing programs to write programs, is fundamental to component generation. Very high level languages (VHLLs) have been shown to be useful for metaprogramming [13, 4]. **perl**[14] is the VHLL used in the work described.

The remainder of the paper contains:

- a description of the problem for which generators were designed and a discussion of alternate approaches.
- a presentation of the generator architecture and design,
- a discussion of the resulting components, **perl** as a generator building language, and the benefits obtained, and
- a description of on-going and future work

2 The Problem

Language processing components like scanners are used in several settings. The components can be found in traditional compilation systems, editors, browsers, file filters, and other applications found in program development environments. Varying requirements are placed upon the components.

The scanner requirements for **scangen** include handling different character sets, varying the storage model (i.e., having the scanner be completely reentrant), changing the target and input language dialects, execution on different computing platforms, and variable amounts of information to be included in the token abstractions.[1] The ability to scan languages other than C or C++ is not a requirement. An additional desirable property, but not a requirement, is to have the scanner's source code (the target language) be either C or C++. Traditional scanner generators like **lex** and **flex**[8] are unsuitable solutions to the problem described. They allow one to change the input language, but little else. **flex** allows more variation than **lex** but it still does not address the requirements well.

Several options may be considered for implementing the required scanners. They are listed here with comments on their suitability for the task.

- <u>modify an existing scanner generator</u> — This is a labor intensive solution. It was rejected because of the amount of programming required and for the anticipated amount of debugging.
- <u>write individual scanners</u> — While it is not very hard to write a C or C++ scanner, doing it several times, with minor modifications is tedious. Even if code were reused a lot, propagating changes and bug fixes to many versions of a program invites problems.
- <u>write one scanner to handle all variabilities</u> — This approach was rejected because the resulting scanner would be too complex, filled with conditionally compiled code that would make it hard to read, and would likely not have the performance demanded for some of the required environments.
- <u>develop a class library for scanners</u> — This approach seemed attractive at first. It was rejected because the approach only allows customization by expansion which makes it difficult to avoid *software bloat* and maintain a small memory footprint where required. This approach also requires scanner modification for use in a C application.
- <u>build a scanner generator that allows for wide variability</u> — This is the approach taken and described in the rest of the paper.

3 Generator Details

3.1 Types of Variability

A component generator must be able to deal with three types of variability: specified constants, simple computations, and parameterized computations. The language used to program the generator should allow one to easily express and implement each type.

A specified constant is a value defined in the specification and given to the generator. A specified constant occurs when a name, such as an identifier or type, is constant in a component but may vary across members in a component family. One uses specified constants for textual substitution during generation. Examples of specified constants in the scanner generator are the file names and extensions of the generated modules.

A simple computation is a computation that occurs once during a single generation. The value computed is constant throughout the generation. A simple computation differs from a specified constant in the way its value is determined. The generator computes the value based upon the specification. Examples of simple computations in the scanner generator are the name of the character type and the values in the `lexemeID` table. The value of the character type depends upon the input character set specified. Values in the lexeme table depend upon the input language, input language dialect, and other characteristics described in the specification.

A parameterized computation is a computation that occurs more than once during a single generation. The value of the computation depends upon values supplied to it by the generator. One uses a parameterized computation to produce program parts that have a standard form with changing values. An example of a parameterized computation is a one that produces a function

[1] For example, some scanners were required to pass complete source information through to the parser so the source program could be recreated. Others were only required to pass the token code and text of identifiers and strings.

declaration. The name of the function, its type, and parameter types and names will vary each time the computation is performed.

3.2 The Scanner Generator Organization

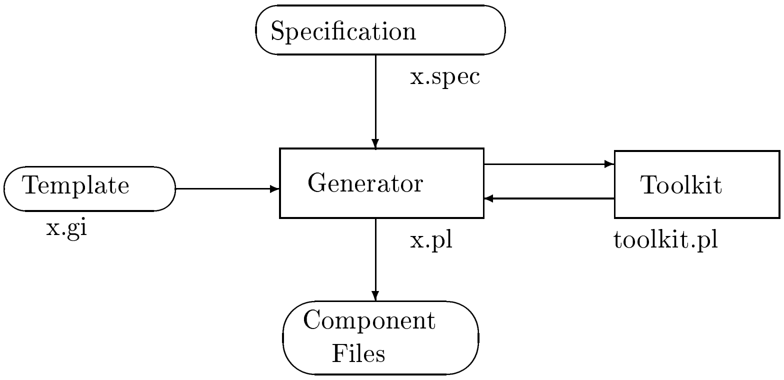

Figure 1: Generator organization.

Several C and C++ scanners were analyzed to identify the common and variable parts. After the analysis, the generator organization shown in Figure 1 was developed. Table 1 briefly describes each of the five components. A more detailed description of each is presented in later sections.

Part	**Contents**
Specification	Specification of the generated component. The specification consists of definitions and values required for generating the variable parts of the component.
Toolkit	General purpose generator functions and procedures. This file is typically limited to specific target languages.
Template	Common source code segments with embedded computations for variable parts.
Generator	Component family specific functions and procedures — the control module for the generator.
Component files	Generator output — C or C++ source code.

Table 1: Contents of a generator's components.

Not all of the generator parts in Figure 1 are required. In some cases there may be no template file or there may be multiple template files. The token module of the scanner generator has no associated template file. The token abstraction is a structure or class containing fields for the token code, string text pointer where applicable, and file position information. Fields may be added or removed, depending upon the application. The generator toolkit automatically generates constructors, destructors, and access functions for structured types if the option is selected in the specification.

3.3 Selecting a Language for Writing the Generator

Metaprogramming is mainly an exercise in text processing where the text is the source code of the resulting programs. In order to implement a source code generator, a VHLL with excellent string manipulation functionality is required. In addition, the VHLL must possess enough computational power to deal with the types of variability described in Section 3.1. In order to support template processing, the language should have the ability to evaluate programs during generation.

Three VHLLs were considered for the scanner generator work: **awk**[7], **perl** (version 4), and **Tcl**[10]. Each is appropriate for programming a generator. **perl** was chosen for several reasons. In addition to its strong string manipulation capability it has parameterized subroutines, an `eval` function, and a rich set of libraries. Most important to the work, **perl** is available on the platforms used (Macintosh and UNIX workstations[2]) and has a debugger.

3.4 The Scanner Specification

The scanner specification is a **perl** source file. The specification consists of a set of assignment statements. The assignments initialize variables with values that completely define the resulting scanner Listing 1 shows a part of a specification for a scanner that is written in ANSI C and accepts ANSI C, including multi-byte characters as described in the ANSI standard for the C language [3].

```
# Information about the language the scanner will accept.
$inputLanguage = "C";                     # C or C++
$inputDialect = "ANSI";                   # ANSI or nonANSI
$charSet = "multibyte";                   # ASCII, multibyte, etc.
$OK8bits = $FALSE;                        # TRUE => 8-bit chars. in identifier
$oldStyleOps = $TRUE;                     # TRUE allows =>, =+, etc.
$allowDollar = $FALSE;                    # TRUE => Allow '$' in identifiers
$maxIDLength = 50;                        # Max. # of chars to keep for
                                          #   identifiers.
###############################################################################
# Information about the target language .
$genTargetLanguage = "C";                 # C or C++
$genTargetDialect = "ANSI";               # ANSI or nonANSI
###############################################################################
# Other variables that control what things get generated and other values.
$generateStrings = $TRUE;                 # TRUE => gen. string abstraction
$generateTokens = $TRUE;                  # TRUE => gen. token abstraction
$generateTests = $TRUE;                   # TRUE => generate test info.
$generateAccessFunctions = TRUE;          # TRUE => generate access functions
                                          #   for data members.
$stringTableSize = 16384;                 # Initial string table size
$stringTableIncrement = 4096;             # How much to grow the string table
                                          #   by when necessary.
$hashSize = 512;                          # Number of hash slots
                                          #   must be power of 2.
$scanErrorFunction = "";                  # User supplied error function.  If
                                          #   an empty string, then ScanError
                                          #   outputs the message.
$tabExpansion = $TRUE;                    # $TRUE => expand tabs for counting
                                          #   column numbers
$tabStop = 8;                             # Number of spaces per tab stop
```

Listing 1. Part of a scanner specification.

3.5 The Generator Toolkit and Templates

The generator toolkit supplies general purpose functions to domain specific generators. There are over forty functions in the toolkit. The functions can be broken down into groups that perform the following tasks:

- formatting source programs,

[2] UNIX is a registered trademark of UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc.

- creating structured and enumerated types, and declarations,
- formatting function declarations, function calls, parameters, and arguments, and
- evaluating templates.

There is one generator toolkit used for the scanner generators. Differences between C and C++ target languages are processed by the toolkit routines. A set of toolkit variables are exposed to the other generator modules. The modules can set these variables to control the toolkit behavior. The variables, $\$genTargetLanguage$ and $\$genTargetDialect$ in Listing 1 are two such variables.

Listing 2 shows a typical generator toolkit function, $\&New$, which generates code to create a new instance of an object. If the target language is C++, the constructor for the class is used. If C is the chosen target language, a function is called which returns a pointer to the appropriate object. The C function name is made up from the name of the type prefixed with the word "New". Consistent style is imposed and maintained by the toolkit code.

```
sub New
{
    local($type, $params) = @_;

    if ($genTargetLanguage eq "C++") {
        if (@_ == 1) { return "new $type"; }
        else { return "new $type($params)"; }
    } else { return "New$type($params)"; }
}
```

Listing 2. The $\&New$ function from the generator toolkit.

An effective template mechanism is one that allows values to be expressed in the template that can be computed at generation time. This requires the generator (and the language in which the language is written) to be able to read embedded statements, evaluate them, and substitute their output for them in the template. The $\&GenEvalFile$ function shown in Listing 3 is the toolkit routine that performs this operation. It works in conjunction with a $\&ReadLine$ and $\&WriteLine$ function to read in a template and output the generated code to the proper file.

Three things are done by $\&GenEvalFile$ on template text lines:

1. If there are no embedded **perl** statements or embedded comments, the text is written to the output with no modification.
2. If there is an embedded comment, convert the comment text to the proper format for the target language and output the comment. Embedded comments in the template begin with '`@@`' and continue until the end of the line.
3. If there is embedded **perl** code, evaluate it and replace the input text with the results of the evaluation for output. Embedded **perl** code is enclosed within '[[' and ']]' delimiters and may be continued on extra lines.

Sample template code is shown in Listing 4. The template is for the $DepositChar$ function in the string table module. The example illustrates variable substitution and replacing a function call with the text it produces (i.e., $\&Params$ to format parameters properly). The last line of the listing causes an empty string to be emitted except when multibyte characters are accepted; in which case the $DepositWChar$ function is emitted.

When a scanner is generated using the specification from Listing 1, the code shown in Listing 5 results when the previous template is evaluated.

```
sub GenEvalFile
{
    local($i, $t, $cmd);

    while (&ReadLine) {
        $i = index($inbuf, "[[");
        while ($i >= $[) {                    # Start of an embedded command.
            $outbuf = substr($inbuf, 0, $i);
            &WriteLine;                       # write out to start of command
            $t = substr($inbuf, $i+2);        # build up the command
            $i = index($t, "]]");
            while ($i < $[) {
                $t .= &ReadLine;              # end not here, keep reading
                $i = index($t, "]]");
            }
            $cmd = substr($t, 0, $i);
            $outbuf = eval($cmd);             # replace the input text
            &WriteLine;
            $inbuf = substr($t, $i+2);
            if (length($inbuf) == 1) { $inbuf = ""; }
            $i = index($inbuf, "[[");
        }
        $i = index($inbuf, "@@");
        if ($i >= $[) {
            $outbuf = substr($inbuf, 0, $i);
            &WriteLine;                       # up to the comment
            $inbuf = substr($inbuf, $i+2);    # remove @@
            chop($inbuf);                     # remove ending newline
            $inbuf = &Comment($inbuf, 0) . $NL;
        }
        $outbuf = $inbuf;
        &WriteLine;
    }
}
```

Listing 3. The *&GenEvalFile* function for evaluating template files.

```
[[$genVoidType]] DepositChar[[&Params("$charType", "c")]]
{
    if (stix >= (stLimit - 4)) {              @@Getting close to limit
        stringTable = (char *)realloc(stringTable,
            (stLimit + [[$stringTableIncrement]]) * sizeof(char));
        stLimit += [[$stringTableIncrement]];
        if (stringTable == NULL) {            @@Error in realloc
            scanAbort = 1;
            ScanError("Error reallocing string table");
            return;
        }
    }
    if (stringTable == NULL)
        return;
    stringTable[stix++] = (char)c;
}
[[$DepositWCharBody]]
```

Listing 4. Template for the *DepositChar* string table function.

```
void DepositChar(wchar_t c)
{
    if (stix >= (stLimit -4)) {                 /* Getting close to limit */
        stringTable = (char *)realloc(stringTable,
            (stLimit + 4096) * sizeof(char));
        stLimit += 4096;
        if (stringTable == NULL) {              /* Error in realloc */
            scanAbort = 1;
            ScanError("Error reallocing string table");
            return;
        }
    }
    if (stringTable == NULL)
        return;
    stringTable[stix++] = (char)c;
}

void DepositWChar(wchar_t c)
{
    char        *cp;
    int         mbCount;

    if (stix >= (stLimit - 4)) {
        stringTable = (char *)realloc(stringTable,
            (stLimit + 4096) * sizeof(char));
        stLimit += 4096;
        if (stringTable == NULL) {
            scanAbort = 1;
            ScanError("Error reallocing string table");
            return;
        }
    }
    if (stringTable == NULL)
        return;
    cp = &(stringTable[stix]);
    mbCount = wctomb(cp, c);
    stix += mbcount;
}
```

Listing 5. Code generated for $DepositChar$.

3.6 Escapes

There are cases where a generator is either insufficient for a specific application but will be close enough to be desirable. One wants to avoid modifying generated code. Once the code has been modified, it must be re-modified each time the component is generated. Escapes are included in a generator to handle such cases by allowing user supplied code to replace generated code.

scangen allows specific functions, like the error reporting function, to be supplied from elsewhere. A default error function is generated which outputs a message on `stderr`. However, if the error function is escaped, the calls to it will remain in the generated code, but the routine must be supplied by the programmer. The name of the error function is stored in the $\$scanErrorFunction$ variable shown in Listing 1.

Complete modules may be omitted from generation, just like functions. In the **scangen** case, each of the string table, I/O, and token modules can be supplied rather than generated.

Languages like **perl** support escapes well. Since they are interpretive functions are only required when they are called. Therefore, if a generator never calls a function it does not have to be supplied. If the feature were not present in the VHLL, default *stub* functions would have to be provided and would have to be overridden when the functionality is required.

3.7 Name Generation

A particularly effective use for generators is to generate names and values for objects in a consistent, understandable manner. In a scanner, token codes must be assigned to each possible entry in the lexicon. Some of these are generic names, like "identifier" while others are specific to one token, like the "!=" operator in C. Generating the names and assigning values to them insures consistency throughout the scanner. This is a particularly error prone operation when manually programming multiple scanners.

```
@archaicOperators = (
"=*", "=/", "=%", "=+", "=-", "=&", "=^", "=|",
"=>", "=<", "=>>", "=<<"
);

###########################################################################
# The following pairs are used to give names to the symbols used
# in making up operators, etc.
#
%tokSymbolNames = (
"_", "under",
"=", "eq",
'"', "quote",
"*", "star",
"%", "pct",
.
.
.
```

Listing 6. Arrays used for naming token codes.

Languages like **perl** that support associative arrays are effective for name generation. The strategy used in **scangen** is to have the base name for each of the token codes stored in one of several arrays. The arrays contain a subset of the possible tokens and each is used as required by the input specification. An additional associative array is used to assign a word to each possible operator character. When a token code name is required, if the base name is a word, the token code prefix, `TC` in this case, is prepended to it. If the base name is an operator, a function is called to turn the operator into a, possibly compound, word and then the token code prefix is prepended.

Listing 6 illustrates the array of archaic C operators and the beginning of the symbol name array. If the token code for the symbol `=*` is required, the $\&MakeTokenWord$ function, shown in Listing 7 is called. It returns the name `TCeqstar`.

```
sub MakeTokenWord
{
    local($symbol) = @_;
    local($i) = 0;
    local($r) = "TC"; # the result

    while ($i < length($symbol)) {
     $r .= $tokSymbolNames{substr($symbol, $i, 1)};
$i++;
    }
    $r;
}
```

Listing 7. $\&MakeTokenWord$, a function for creating token names.

4 Results

Results of the scanner generator work is described in the following sections. A couple of versions of the generator have been produced which have generated dozens of scanners. Several areas of consideration are discussed.

4.1 Productivity Improvement

One way to determine the usefulness of a generator is to measure the ratio of generator code to generated code for one component. For instance, generating programs to input a description of a deterministic finite automaton (DFA) and produce a C++ program that represents the DFA has yielded a ratio of generator code to generated C++ program code of about $1:4$ [9]. The scanner generator, not including the code for the generator toolkit, yields a ratio of about $4:3$.

If lines of code is the only metric used, one would judge the scanner generator a failure. However, many scanners can be generated by the generator. If, for instance, eight different scanners are generated by the scanner generator we have a ratio of $1:6$ (i.e., $4:8*3$).

If one were to hand code multiple versions of a scanner, pieces of code would be used in a cut-and-paste fashion. However, multiple versions of the scanner would have to be maintained. The duplicated maintenance work is eliminated with the generation approach.

One of the biggest productivity gains came in regression testing. The output of a component generator is a source program. Modifying the generator to produce new members of the component family should not change previously generated source programs in any substantial way. Changes to source programs are easy to locate and the detection takes little time compared with re-running a test suite.[3] Since there is no conditionally compiled code in the generated source programs, features that are not appropriate to a particular scanner are never part of its source code.

4.2 Performance

The goal was to produce scanners that were as efficient as hand coded ones. Since the code generated is almost identical to the hand coded scanners used as models, the goal was achieved.

A scanner generated by **scangen** has been compared to a scanner generated by the **lex** and **flex** scanner generators. The results are shown in Table 2. All scanners were generated on a Sun SPARCstation ELC under SunOS 4.1.3. The **lex** version is the one shipped with the operating system. **flex** version 2.3 was used for the comparison. Three versions of a **flex** scanner were generated. These represented the extremes in the space-time tradeoff spectrum and a middle of the road version. The **scangen** scanner used non-generated token and I/O modules. This was done in order to make sure similar scanners were being compared. Each scanner used the standard input stream and returned a simple integer token code for each token recognized. The times shown are the average over several runs of scanning an approximately 50,000 line C++ source file. Times are shown in seconds. The program size is the sum of `text`, `data`, and `bss` as shown by the **size** program.

4.3 Maintainability

Multiple versions of the scanner generator have been produced. This is not a desirable feature. One would like to maintain just one generator. However, the generator's complexity grows as quickly as any comparably sized program. Therefore, while the maintenance effort is not any greater (and is, in fact spread out over the number of components generated), maintenance is not, however, any easier with the type of generator described.

Generators need to be expanded and contracted, just like any other family of programs. The generators described are single points that generate a family of components. A more desirable

[3] It takes about five seconds to generate a scanner with the scanner generator and five additional seconds to compare the generated source modules with previously generated versions. It takes several minutes to run a typical test suite for C scanners.

Scanner	Real time	User time	Program size
scangen	4.8	4.5	22888
lex	10.6	10.3	30368
flex	5.4	5.2	16384
flex -C	5.0	4.7	24576
flex -Cf	3.3	3.1	106496

Table 2: **scangen** scanner performance vs. standard generators.

architecture is one that has a family of generators, each of which generates a family of related components. This is further discussed in Section 5.

Use of the **scangen** generator beyond the initial implementation led to a major modification. The original generator that generated four modules was broken up into four generators, each generating one module. Each generator uses the same input specification.

4.4 perl as a Generator Implementation Language

For most things, **perl** was a good choice for the generator implementation language. It allowed changes to be implemented and tested quickly and the debugger was invaluable.

perl is a big language which makes mastery difficult. The documentation to date is barely adequate. A lot of time was spent trying to figure out how to get something to work properly. In most cases there was more than one way of getting the job done. The advantages and disadvantages of each were not clear.

Since metaprogramming tasks are mostly string manipulation, **perl** was more than adequate for the work. One would like to write code segments in the generator as close as possible to the way they appear in the generated code. Variable substitution in strings helps make this possible. However, in order to handle strings generated from parameterized computations, a desirable feature of an implementation language would be to allow function calls to be replaced with the result of the function inside of strings (or provide a special string type that would allow this).

For example, in the token module, the following line of code appears:

```
$scanReserve .= "    Reserve( " . &str($cppKeywords[$i]) . ", TC$cppKeywords[$i]);\n";
```

A more desirable line of code is:

```
$scanReserve .= "    Reserve( &str($cppKeywords[$i]), TC$cppKeywords[$i]);\n";
```

5 Future Work

The problem of generator maintainability, extensibility, and contraction was mentioned in Section 4.3. Current work is focusing on being able to formally describe the generated family of components. Such descriptions require common and variable parts to be expressed in a way that generators can be constructed and maintained.

The current generator model is shown in Figure 2. One generator is constructed to produce several members of a family. All possible variabilities are handled by the generator. So, for example, if there is no requirement for reentrancy, no reentrant specific code is produced. However, the generator has several places where decisions about whether to generate reentrant code must be made.

A system for creating domain specific languages (DSLs) to describe the components and a generator framework are being explored. The framework allows one to compose an appropriate generator given a specification for a family of components. In such a system, generators for a domain are created by composing them from compatible parts. Each generator in the family of generators works on an unique set of variabilities.

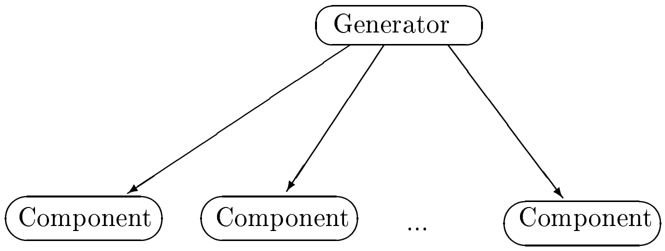

Figure 2: Current generator model.

Figure 3 shows a model for a family of generators. Heavy arrows show generator extension (i.e., sharing of variabilities). In the example, G_4 is an extension of G_2 and G_6 is a shared extension of G_3 and G_5.

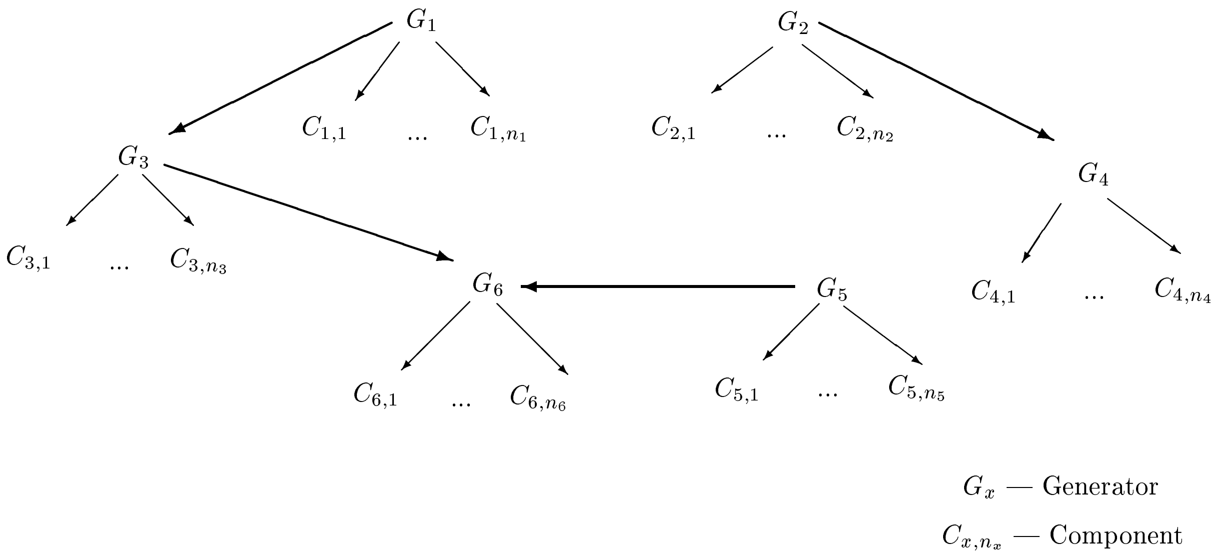

Figure 3: A family of generators.

The desired system allows the generator producer to work at a high level, using language appropriate for the domain. The system performs the necessary semantic analysis and bookkeeping to construct generators from the appropriate parts. Each generator functions like the generators described in this paper. At least one VHLL, possibly more, is used in the system. The generators are written in a VHLL. The framework for the system will most likely be written in a VHLL as well.

6 Availability

The original **scangen** sources may be obtained as a **shar** file from the author. These are the source files used in the M.S. thesis, which may also be obtained from the author as a compressed, encoded PostScript file. The author may be contacted at pollice@centerline.com or gpollice@cs.uml.edu.

7 Acknowledgments

I would like to thank CenterLine Software, Inc. for providing hardware and software resources used to develop and refine the **scangen** generators. Thanks also go to Bill McKeeman and David Weiss for providing feedback on the work. Their comments have and shaped continue to shape the direction of my work.

8 Biographical Information

Gary F. Pollice is a software engineer at CenterLine Software, Inc., Cambridge, MA. He is also in the doctoral program at the University of Massachusetts, Lowell. He received a B.A. in mathematics from Rutgers, the State University of New Jersey, and a M.S. in computer science from U. Massachusetts, Lowell. His research interests are software reuse, compiler technology, and software engineering.

References

[1] Reuse-driven software process guidebook. Technical Report SPC-92019-CMC, Software Productivity Consortium, Herndon, VA, November 1993.

[2] The SDDR design concept. Mosaic pages, 1994.

[3] American National Standards Institute. *American National Standard for Information Systems — Programming Language C*, February 1990. Doc. #X3J11/90-013.

[4] Jon Bentley. Template-driven programming. *Unix Review*, 12(4):79–88, April 1994.

[5] J. Craig Cleaveland. Building application generators. *IEEE Software*, pages 25–33, July 1988.

[6] Kyo C. Kang, Sholom Cohen, Robert Holibaugh, James Perry, and A. Spencer Petersoft. A reuse-based software development methodology. Technical Report CMU/SEI-92-SR-22, Software Engineering Institute, Pittsburgh, PA 15213, January 1992.

[7] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[8] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* O'Reilly & Associates, Inc., Sebastopol, CA, 1990.

[9] W. M. McKeeman. Personal Correspondence, 1993. Electronic mail message.

[10] John K. Ousterhout. *Tcl and the Tk Toolkit.* Professional Computing Series. Addison-Wesley, Reading, MA, 1994.

[11] David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[12] Gary F. Pollice. Component generation as a software reuse technique illustrated with a C scanner generator. Master's thesis, U. Massachusetts, Lowell, 1994.

[13] Christopher J. Van Wyk. AWK as glue for programs. *Software—Practice and Experience*, 16(4):369–388, April 1986.

[14] Larry Wall and Randal L. Schwartz. *Programming perl.* O'Reilly & Associates, Inc., Sebastopol, CA, 1990.