Generative Programming and Active Libraries
Extended Abstract

Krzysztof Czarnecki[1], Ulrich Eisenecker[2], Robert Glück[3],
David Vandevoorde[4], and Todd Veldhuizen[5]

[1] DaimlerChrysler AG (Research and Technology, Ulm) czarnecki@acm.org
[2] Fachhochschule Heidelberg Ulrich.Eisenecker@t-online.de
[3] University of Copenhagen (Dept. of Computer Science) glueck@diku.dk
[4] Edison Design Group daveed@vandevoorde.com
[5] Indiana University (Computer Science Dept.) tveldhui@acm.org

Abstract. We describe *generative programming*, an approach to generating customized programming components or systems, and *active libraries*, which are based on this approach. In contrast to conventional libraries, active libraries may contain metaprograms that implement domain-specific code generation, optimizations, debugging, profiling and testing. Several working examples (Blitz++, GMCL, Xroma) are presented to illustrate the potential of active libraries. We discuss relevant implementation technologies.

1 Introduction

The main goal of generic programming is to improve reusability by providing parameterized components which can be *instantiated* for different choices of parameters. The C++ Standard Template Library [24] is remarkable because generic algorithms can be orthogonally combined with generic container representations. The aims of *generative programming* resemble those of generic programming but differ in several important aspects that can enhance the power of generic programming.

Generative Programming (Sect. 2) is about modeling families of software systems by software entities such that, given a particular requirements specification, a highly customized and optimized instance of that family can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge (see [6,9]). Generative Programming involves all phases of the software development process, from the specification of the abstract software entities, to the design and implementation of generative software systems, and the mapping of these onto hardware within constraints (such as space and speed).

We refer to libraries which apply concepts of generative programming as *Active Libraries* (Sect. 3). Active libraries require implementation technologies beyond what is provided by traditional compilers (Sect. 3.2). To illustrate these ideas, we present working examples: Blitz++ and the Generative Matrix Computation Library (GMCL) (Sect. 4), which are C++ libraries that generate cus-

M. Jazayeri, R. Loos, D. Musser (Eds.): Generic Programming '98, LNCS 1766, pp. 25–39, 2000.

tomized components; the Tau package (Sect. 5), an extensible profiling tool; and the Xroma system (Sect. 6) for extensible compilation.

2 Generative Programming: Goals and Principles

The goal of modern programming is often to design *domain-specific abstractions* and implement these abstractions using some programming language. In providing these abstractions, we have to deal with two major problems: (1) domain-specific knowledge may get lost in the implementation because there exists a semantic gap between domain-specific abstractions and features offered by a programming language; (2) flexibility and generality in the design may incur considerable performance penalties at runtime.

Generative programming (GP) is an approach to generating customized components and systems. The goals are to (a) decrease the conceptual gap between domain concepts and program code (known as achieving high *intentionality*), (b) achieve high reusability and adaptability, (c) simplify managing many variants of intermediate and end-products, and (d) increase efficiency (both in space and execution time). To meet these goals, GP deploys several principles:

Parameterization of differences: As in generic programming, parameterization allows us to represent—in a compact way—families of components (i.e. components with many commonalities).

Analysis and modeling of dependencies and interactions: Not all parameter value combinations are valid, and the values of some parameters may imply the values of some other parameters. These dependencies are referred to as *horizontal configuration knowledge*, since they occur between parameters at one level of abstraction.

Overhead elimination and domain-specific optimizations: By generating components statically (at compile time), much of the overhead due to unused code, run-time checks and unnecessary levels of indirection may be eliminated. Complicated domain-specific optimizations may also be performed (for example, loop transformations for scientific codes).

Separating problem and solution spaces: The problem space consists of the application-oriented concepts and features in terms of which application designers would like to express their needs, whereas the solution space contains elementary, reusable implementation components (e.g., generic components) which may be combined, analyzed and transformed by metaprograms in order to produce the desired software entity. Configuration knowledge is used to map from the problem space to the solution space. Configuration knowledge may capture specific information about default settings and parameter dependencies, illegal feature combinations, specific construction and optimization rules. The distinction between problem space, solution space and configuration knowledge allows the specification of systems by abstract features (while generic programming expects parameters that instantiate a concrete component).

Separation of concerns: This term, coined by Dijkstra, refers to the importance of dealing with one important issue at a time. To avoid program code which

deals with many issues simultaneously, generative programming tries to separate each issue into a distinct set of code. These pieces of code are then combined to generate a needed component. This idea is borrowed from Aspect-Oriented Programming [21].

2.1 Towards Generative Programming

There are three other programming paradigms which have goals similar to those of Generative Programming: Generic Programming, Domain-Specific Languages (DSLs), and Aspect-Oriented Programming (AOP). Generative Programming is broader in scope than these, but borrows important ideas from each:

Generic Programming may be summarized as "reuse through parameterization." Generic programming allows components which are extensively customizable, yet retain the efficiency of statically configured code [24]. This technique can eliminate dependencies between types and algorithms that are not necessary. For example, iterators allow generic algorithms which work efficiently on both dense and sparse matrices [28].

However, generic programming limits code generation to substituting concrete types for generic type parameters, and welding together pre-existing fragments of code in a fixed pattern. It does not allow generation of completely new code. *Generative* programming is more general because it provides automatic configuration of generic components from abstract specifications, and a more powerful form of parameterization. For example, automatic configuration allows abstract parameters such as an optimization flag (e.g., *speed*, *space*, or *accuracy*). Such parameters map to configurations of implementation components rather than just single parameter components.

Domain-Specific Languages (DSLs) provide specialized language features that increase the abstraction level for a particular problem domain; they allow users to work closely with domain concepts, at the cost of language generality. Domain-specific languages range from widely-used languages for numerical and symbolic computation (e.g., Mathematica) to less well-known languages for telephone switches and financial calculations (to name just a few application domains). DSLs are able to perform domain-specific optimizations and error checking. On the other hand, traditional DSLs typically lack support for generic programming. Shortcomings of traditional DSLs may be overcome by embedding DSLs in existing languages, as advocated in [18].

Aspect-Oriented Programming. Most current programming methods and notations concentrate on finding and composing functional units, which are usually expressed as objects, modules and procedures. However, several properties such as error handling and synchronization cannot be expressed cleanly and locally using current (e.g., object-oriented) notations and languages. Instead, they

are expressed by small code fragments scattered throughout several functional components.

Aspect-Oriented Programming (AOP) [21] decomposes problems into functional units and *aspects* (such as error handling and synchronization). In an AOP system, components and aspects are *woven* together to obtain a system implementation that contains an intertwined mixture of aspects and components (called *tangled* code). Weaving can be performed at compile time (e.g., using a compiler or a preprocessor) or at runtime (e.g., using dynamic reflection). In any case, weaving requires some form of metaprogramming (see [4]). Generative programming has a wider scope that includes automatic configuration and generic programming techniques, and provides new ways of interacting with the compiler and development environment.

Putting It Together: Generative Programming. The concept of generative programming encompasses properties of the previous three paradigms, as well as some additional techniques to achieve the goals listed in Sect. 2:

- DSL techniques are used to improve intentionality of program code, and to enable domain-specific optimizations and error checking.
- AOP techniques are used to achieve *separation of concerns* by isolating aspects from functional components.
- Generic Programming techniques are used to parameterize over types, and iterators are used to separate out data storage and traversal aspects.
- Configuration knowledge is used to map between the problem space and solution space. Different parts of the configuration knowledge can be used at different times in different contexts (e.g., compile time or runtime or both).

3 Active Libraries

As noted in the previous section, Generative Programming requires metaprogramming for weaving and automatic configuration. Supporting domain-specific notations may require syntactic extensions and non-textual and interactive representations. Libraries based on Generative Programming ideas thus need both implementation code, and *metacode* which can implement syntax extensions and rendering of textual and non-textual program representations, perform code generation, and apply domain-specific optimizations.[1]

Active libraries are not passive collections of routines or objects, as are traditional libraries, but take an active role in generating code. Active libraries provide abstractions and can optimize those abstractions themselves. They may generate components, specialize algorithms, optimize code, automatically configure and tune themselves for a target machine, check source code for correctness, and report compile-time, domain-specific errors and warnings. They may also describe

[1] We are stretching the traditional meaning of *library*; we do not imply a specific packaging technology.

themselves to tools such as profilers and debuggers in an intelligible way or provide domain-specific debugging, profiling, and testing code themselves. Finally, they may contain code for rendering domain-specific textual and non-textual program representations and for interacting with such representations.

This perspective forces us to redefine the conventional interaction between compilers, libraries, and applications. Active Libraries may be viewed as knowledgeable agents, which interact with each other to produce concrete components. Such agents need infrastructure supporting communication between agents, code generation and transformation, and interaction with the programmer.

3.1 Types of Active Libraries

Active libraries possess different levels of sophistication, characterized by the type of metaprogramming used: (i) active libraries that extend a compiler, (ii) active libraries that extend the environment to provide domain-specific tool support, and (iii) active libraries that contain *metacode* to analyze and transform the domain-specific concepts at different times and in different contexts.

Extending a Compiler. Active libraries may extend a compiler by providing domain-specific abstractions with automatic means of producing optimized program code. They may compose and specialize algorithms, automatically tune code for a target machine and instrument code. Both Blitz++ and the Generative Matrix Computation Library are examples of libraries that extend a compiler (Sect. 4).

Domain-Specific Tool Support. Active libraries may extend the programming environment to provide domain-specific debugging support, domain-specific profiling and code analysis capabilities, and so on. An example is the interaction between Tau and Blitz++ (Sect. 5). An example of a programming platform which supports these ideas as well as Extended Metaprogramming described below is the Intentional Programming (IP) system [29].

Extended Metaprogramming. Active Libraries may contain *metacode* which can be executed to compile, optimize, adapt, debug, analyze, visualize, and edit the domain-specific abstractions. Active libraries may generate different code depending on the deployment context; for example, they may query the hardware and operating system about their architecture. Furthermore, the same metacode may be used at different times and in different contexts; for example, based on the compile-time knowledge of some context properties which remain stable during runtime, some metacode may be used to perform optimizations at compile time and other metacode may be injected into the application to allow for optimization and reconfiguration at runtime.

3.2 Implementation Technologies

Active Libraries require languages and techniques which open up the programming environment. Issues that need to be addressed when constructing active libraries are language support, transformation and analysis, and tool interfacing. We survey these issues in this section. We list those technologies which we believe are most relevant at the time of this writing and which have been put to good use in existing generators.

Language Support. A key issue in the construction of active libraries is support for generative tasks by the implementation language. Generative programming is a novel concept and existing languages are not well-equipped for it.

C++ Templates The C++ language includes some compile-time processing abilities quite by accident, as a byproduct of template instantiation. Nested templates allow compile-time data structures to be created and manipulated, encoded as types; this is the basis of the expression templates technique [33]. The template metaprogram technique [34] exploits the template mechanism to perform arbitrary computations at compile time; these "metaprograms" can perform code generation by selectively inlining code as the "metaprogram" is executed. This technique has proven a powerful way to write code generators for C++.

Extensible compilation and Metalevel processing In metalevel processing systems, library writers are given the ability to directly manipulate language constructs. They can analyze and transform syntax trees, and generate new source code at compile time. The MPC++ metalevel architecture system [19] provides this capability for the C++ language. MPC++ even allows library developers to extend the syntax of the language in certain ways (for example, adding new keywords). Other examples of metalevel processing systems are Open C++ [2], Magik [10], and Xroma (Sect. 6). An important differentiating factor is whether the metalevel processing system is implemented as a pre-processor, an open compiler, or an extensible programming environment (e.g., IP [29]). A potential disadvantage of metalevel processing systems is the complexity of code which one must write: modern languages have complicated syntax trees, and so code which manipulates these trees tends to be complex as well. The situation can be improved by replacing the direct manipulation of syntax trees with a declarative metalanguage for manipulating code. For example, the FOG system [37] provides such a metalanguage for C++.

Meta-languages Program generation is a specific metacomputation task that stretches the capabilities of usual programming languages. Few programming languages support code manipulation and transformation. Design and implementation of such generative programming languages requires novel concepts (e.g., type systems [31]). An important concept that stems from the area of partial evaluation is that of two-level (or more generally, multi-level) programming languages. Two-level languages contain static code (which is evaluated

at compile-time) and dynamic code (which is compiled, and later executed at run-time). Multi-level languages [15] can provide a simpler approach to writing program generators (see for example, the Catacomb system [30]).

Program Transformation. Another key issue in the construction of active libraries is the use of program analysis and transformation to optimize and weave the implementation code. Exactly *when* these activities take place depends on the binding-times of the library components and the abstract specifications. Three fundamental operations are the essence of a wide spectrum of automatic transformation methods [16]: program specialization, program composition, and program inversion. Here we shortly discuss program specialization, program composition, and the transformation of programs at run-time.

Program Specialization provides means to tailor generic and highly parameterized components to specific needs and applications. One of the best developed specialization techniques is *partial evaluation* [1,8,20]. An extensive theory and literature on specialization and code generation was developed in this field.

An important discovery was that the concept of *generating extensions* [12] unifies a wide class of apparently different program generators. Examples include parsing, translation, theorem proving, and pattern matching [14]. Through partial evaluation, components which handle variability at run-time can be transformed into *component generators* (or *generating extensions* in the terminology of the field) which handle variability at compile-time [23]. The *Futamura projections* [13] provide the theoretical cornerstone for this technique.

In some cases, such transformations avoid the need for library developers to work with complex meta-level processing systems. Automatic tools for turning a general component into a component generator (or *generating extension*) now exist for various programming languages such as Prolog, Scheme, and C (see [20]).

Program Composition The construction of software by sharing and combining existing implementation components is a main activity when producing customized software. Unfortunately, program hierarchies and modularity do not come for free: they add intermediate data structures, redundant computations, interface code and error checking. Program composition techniques (e.g., [32,36]) can remove such redundancies, and allow fusing components without paying an unacceptably high price.

Runtime Code Generation systems allow libraries to generate customized code at run-time. This makes it possible to perform optimizations which depend on information not available until run-time, for example, the structure of a sparse matrix or the number of processors in a parallel application. Examples of such systems which generate native code are ‘C (Tick-C) [11,26], Fabius [22], Tempo-C [3], and DyC [17]. Code generation speeds as high as 6 cycles per generated instruction have been achieved. Runtime code modification can also be achieved

using dynamic reflection facilities available in languages such as Smalltalk and CLOS.

Interfacing. The concept of active libraries requires us to redefine the interaction of traditional programming tools. We mentioned self-instrumentation of active libraries, which allows new possibilities for profiling, debugging, and program analysis.

Extensible Tools To provide domain-specific support in programming environments, we need tools that provide hooks for libraries to define customized debugging support, profiling, etc. Examples of such tools are Tau (Sect. 5) and the Intentional Programming system [29].

4 Examples: Active Library Extending a Compiler

Recently there have been several projects in scientific computing which fit the description of active libraries. This trend is a result of two main factors: a desire for high-level abstractions which closely model the problem domain, and the inability of traditional compilers to optimize such abstractions. We describe two such libraries here.

Blitz++ The Blitz++ library [35] provides array objects for C++ similar to those in Fortran 90. The largest performance problem for arrays in C++ has been temporaries which result from overloaded operators. Blitz++ solves this problem using the *expression templates* technique [33], which allows it to generate custom evaluation kernels for array expressions. The library performs many loop transformations (tiling, reordering, collapsing, unit stride optimizations, etc.) which have previously been the responsibility of optimizing compilers. Blitz++ generates different code depending on the target architecture. The *template metaprogram* technique [34] is used to generate specialized algorithms for operations on small vectors and matrices.

The Generative Matrix Computation Library (GMCL) is able to generate matrix components with a selected combination of features such as element types (real numbers), density (dense and sparse), storage formats (row- and column-wise, several sparse formats), memory allocation (dynamic and static), error checking (bounds, compatibility, memory allocation), and operations (addition, subtraction, multiplication). Only some combinations of these features are valid; these are specified by a configuration DSL. A configuration DSL has the form of a parameter space, where the component to be configured has a number of parameters and the parameter values can be parameterized themselves. A particular configuration expression is translated into a concrete configuration of generic components. Currently there exist two implementations of the GMCL, one in the Intentional Programming system and one in C++

(see [25,4] for details). The C++ GMCL makes widespread use of expression templates [33], generative programming idioms in C++ [9,4], and many template metaprogramming facilities, e.g., control structures for static metaprogramming [5]. Figure 1 demonstrates the translation of a matrix configuration expression into a configuration of generic components in C++ GMCL. The translation is performed by a template metaprogram. GMCL contains another generator for generating efficient implementations of matrix expressions (e.g., "(A+B)*(C+D)"). This generator reads out the properties of the operands from their configuration repositories (Figure 1).

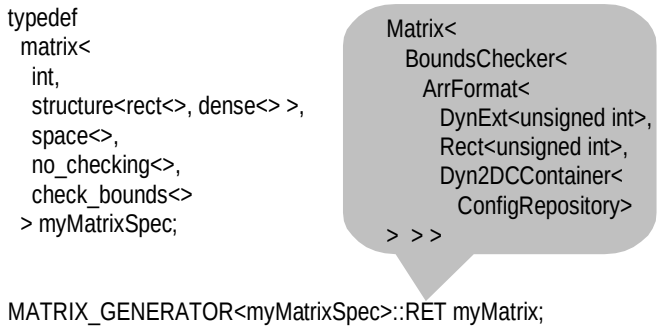

Fig. 1. Sample C++ code specifying a rectangular dense matrix with bounds checking. The generated matrix type is shown in the gray area

The C++ implementation of the matrix component [25] comprises 7500 lines of C++ code (6000 lines for the configuration generator and the matrix components and 1500 lines for the operations). The matrix configuration DSL covers more than 1840 different kinds of matrices. Despite the large number of supported matrix variants, the performance of the generated code is comparable with the performance of manually coded variants. This is achieved through exclusive use of static binding, which is often combined with inlining.

5 Example: Active Library for Domain-Specific Tool Support

Tau [27] is a package for tuning and analysis of parallel programs. Unlike most profilers, Tau allows libraries to instrument themselves. Self-instrumentation solves some important problems associated with tracing and profiling large libraries. Libraries often have two (or more) layers: a user-level layer, and layers of

internal implementation routines. Automatic instrumentation mixes these layers, which may result in a swamp of hard-to-interpret information. The problem is compounded when template libraries (with their long symbol names and many template instances) are used. With self-instrumentation such as Tau offers, active libraries can present themselves to profilers and tracers in an understandable way: time spent in internal routines can be properly attributed to the responsible interface routines, and complex symbol names (especially template routines) can be rewritten to conform to a user's view of the library. For example, Blitz++ uses pretty-printing to describe expression templates kernels to Tau. When users profile applications, they see the time spent in array expressions such as "A=B+C+D", rather than incomprehensible template types and runtime library routines.

6 Xroma: Extensible Translation

Xroma (pronounced Chroma) is a system being developed by one of us (Vandevoorde) specifically to support the concept of active libraries. We describe it here to illustrate in detail an extensible compilation system and metalevel processing.

The Xroma system provides an API for manipulating syntax trees, and a framework which allows library developers to extend the compilation process (Figure 2). Syntax trees which can be manipulated are called *Xromazene*. The parser of the Xroma language produces well-specified Xromazene that is fed to the Xroma translation framework. The grammar of the Xroma language is static: it is not possible to add new syntax structures. Instead, the translation is extensible by defining new Xromazene transformations. The Xroma language describes both the transformations and the code upon which they act.

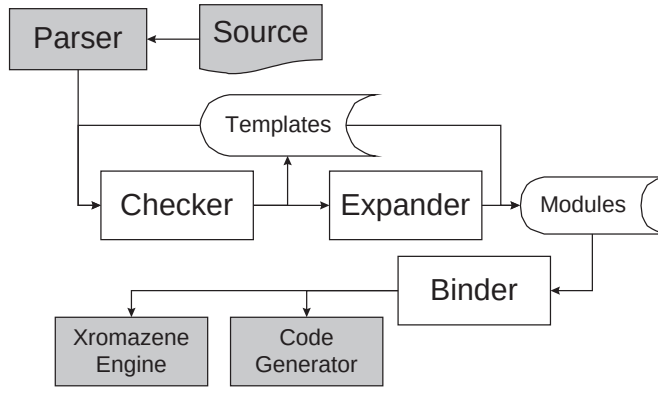

Fig. 2. General organization of Xroma

Libraries in Xroma are composed of *modules*, which are collections of types, procedures and templates. Modules may contain *active components* which are able to generate or transform code.

Xroma Components. The Xroma system exposes several components: a *checker*, an *expander*, a module manager, a template manager and a *binder*. Active components can attach themselves to these Xroma components. The output of the binder is a Xromazene file that can be interpreted on a Xromazene engine or fed to a traditional optimizing code generator. The binder can activate modules e.g., to implement domain-specific global optimizations. The checker's role is to verify semantic consistency of types, procedures and templates. If any of these elements has an active component attached, that component can take over control of checking. For example, it could relax the type-matching requirements between a call site and the available procedure declarations (overload set), or ensure early enforcement of template instantiation assumptions. The expander is a second pass that must reduce application-specific nodes in the syntax tree to the generic set of nodes understood by the binder. Often, an active component does not want to take control of the translation of a whole procedure, type or template. Instead, it is usually only desirable to intercept specific constructs. The Xroma system allows these events to be captured by active libraries at checker or expander time: type, procedure and template definitions; call-sites (before and after overload resolution); object definitions (constructor calls), and template instantiation.

Xroma Example. The following example demonstrates a miniature active library that enforces a constraint on template type-arguments: the type for which the template is instantiated should not be a reference type. Two modules are presented: the first one makes use of the active library, and the second implements the library. The first module defines a template type `Value` whose parameter may only be a non-reference type; this is enforced by the `NotRef` component (defined later):

```
module["program"] AnnotationDemo {      // 1
   synonym module XD = XromasomeDemo;   // 2

   // Declare the Value type            // 3
   export template[type T]              // 4
   type[XD::NotRef(0)] Value {          // 5 Annotation-brackets
      export proc init(ref rdonly T);   // 6
      var T v_;                         // 7
   }                                    // 8

   // Example uses of Value:            // 9
   proc init() {                        // 10 Module initialization
      var Value[int[32]] a;             // 11 Okay
      ref Value[int[32]] b;             // 12 Okay
```

```
      var Value[ref int[32]] c;          // 13 Error!
   }                                     // 14
}                                        // 15 End program module
```

The keyword `type` (line 5) is similar to the C++ keyword `class`. Square brackets are used to denote template parameters (line 4), template arguments (lines 11-13) and annotations. Annotations (line 5) allow the Xroma programmer to attach active components to types and procedures. In the example above, this component is `NotRef(0)` where "0" indicates that the first template argument should be checked. The Xroma translation system attaches this annotation to the template and executes the associated active component every time an attempt is made to instantiate the template (lines 11-13). The special procedure `init()` (lines 6,10) corresponds to object constructors and module initializers.

Here is the definition of the `NotRef` module:

```
module XromasomeDemo {                                  // 1
   synonym module XT = Xroma::Translator;               // 2

   export type NotRef extends XT::Xromasome {           // 3
      export proc init(int[32] a) a_(a) {}              // 4
      virtual phase()->(XT::Phase request)              // 5
         { request.init(XT::kGenerate); }               // 6
      virtual generate(ref XT::Context n);              // 7
      var int[32] a_; // Template argument number      // 8
   }                                                    // 9

   virtual NotRef:generate(ref XT::Context n) {         // 10
      ref t_spec = convert[XT::TemplSpec](n.node());    // 11
      if t_spec.arg(a_).typenode()==null {              // 12
         XT::fatal(&n, "Unexpected non-type arg");      // 13
      } else if t_spec.arg(a_).typenode().is_ref() {    // 14
         XT::fatal(&n, "Unexpected ref type");          // 15
      } else { XT::Xromasome::generate(&n); }           // 16 Call base
   }                                                    // 17
} // End module XromasomeDemo                          // 18
```

Active components are implemented as objects that respond to a Xromasome interface provided by the framework. When the active component attaches itself to some aspect of the Xroma system, the method `phase()` (line 5) is invoked to determine at which stage the Xromasome wishes to take control. When that stage is encountered for the construct to which the Xromasome was attached, another method `generate()` (line 10) for template instantiation is invoked with the current context of translation. In our example the template argument is retrieved from the Xromazene, and after verification that it is a non-reference type argument, the default instantiation behavior accessible through the Xromasome interface is called.

7 Conclusion and Future Work

We presented an approach to generating software components which we call Generative Programming, and described Active Libraries, which implement this approach. Our approach has several advantages over conventional, passive libraries of procedures or objects: it allows the implementation of highly-parameterized and domain-specific abstractions without inefficiency. Even though we presented examples from the area of scientific computing and extensible translation, our approach should be applicable to other problem domains (business components, network applications, operating systems, etc.).

In summary, the ability to couple high-level abstractions with code transformation and implementation weaving enables the development of highly intentional, adaptable and reusable software for generating highly optimized applications or components. However, it is also clear that more work will be required, including research on generative analysis and design methods (e.g., the use of Domain Analysis and OO methods [7]), development of metrics and testing techniques for generative programs, further research on integrating ideas from related areas (e.g., domain-specific languages, generators, automatic program transformation and optimization of programs, metaobject protocols, and Domain Engineering), and, finally, the development of industrial strength generative programming tools.

References

1. D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, Amsterdam, 1988.
2. S. Chiba. A Metaobject Protocol for C++. In *OOPSLA'95*, pages 285–299, 1995.
3. C. Consel, L. Hornof, F. Nöel, J. Noyé, and E. N. Volanschi. A uniform approach for compile-time and run-time specialization. In Danvy et al. [8], pages 54–72.
4. K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. PhD Thesis*. Technical University of Ilmenau, Ilmenau, 1998.
5. K. Czarnecki and U. Eisenecker. Meta-control structures for template metaprogramming. `http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm`.
6. K. Czarnecki and U. Eisenecker. Components and generative programming. In O. Nierstrasz, editor, *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE 99, Toulouse, Frankreich, September 1999)*. Springer-Verlag, 1999.
7. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley Longman, 1999. (to appear).
8. O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
9. U. Eisenecker. Generative Programming (GP) with C++. In H. Mössenböck, editor, *Modular Programming Languages*, volume 1024, pages 351–365. Springer-Verlag, 1997.

10. D. R. Engler. Incorporating application semantics and control into compilation. In *USENIX Conference on Domain-Specific Languages (DSL'97)*, October 15–17, 1997.
11. D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *POPL'96*, pages 131–144, 1996.
12. A. P. Ershov. On the essence of compilation. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.
13. Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
14. Y. Futamura. Partial computation of programs. In E. Goto, K. Furukawa, R. Nakajima, I. Nakata, and A. Yonezawa, editors, *RIMS Symposia on Software Science and Engineering*, volume 147 of *Lecture Notes in Computer Science*, pages 1–35, Kyoto, Japan, 1983. Springer-Verlag.
15. R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, 1997.
16. R. Glück and A. V. Klimov. Metacomputation as a tool for formal linguistic modeling. In R.Trappl, editor, *Cybernetics and Systems '94*, volume 2, pages 1563–1570, Singapore, 1994. World Scientific.
17. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-directed run-time specialization in C. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, pages 163–178. ACM, June 1997.
18. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196–196, Dec. 1996.
19. Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota. Design and implementation of metalevel architecture in C++ – MPC++ approach. In *Reflection'96*, 1996.
20. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).
21. G. Kiczales and et al. Home page of the Aspect-Oriented Programming Project. `http://www.parc.xerox.com/aop/`.
22. M. Leone and P. Lee. Lightweight run-time code generation. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106. Technical Report 94/9, Department of Computer Science, University of Melbourne, June 1994.
23. R. Marlet, S. Thibault, and C. Consel. Mapping software architectures to efficient implementations via partial evaluation. In *Conference on Automated Software Engineering*, pages 183–192, Lake Tahoe, Nevada, 1997. IEEE Computer Society.
24. D. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Reading (MA), USA, 1996.
25. T. Neubert. *Anwendung von generativen Programmiertechniken am Beispiel der Matrixalgebra*. Technische Universitet Chemnitz, 1998. Diplomarbeit.
26. M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *PLDI'97*, 1996.
27. S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable profiling and tracing for parallel, scientific applications using C++. In *Proceedings of SPDT'98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 134–145, August, 1998.

28. J. G. Siek and A. Lumsdaine. *Modern Software Tools in Scientific Computing*, chapter A Modern Framework for Portable High Performance Numerical Linear Algebra. Birkhauser, 1999.
29. C. Simonyi and et. al. Home page of the Intentional Programming Project. `http://www.research.microsoft.com/research/ip/`.
30. J. Stichnoth and T. Gross. Code composition as an implementation language for compilers. In *USENIX Conference on Domain-Specific Languages*, 1997.
31. W. Taha, Z.-E.-A. Benaissa, and T. Sheard. Multi-stage programming: axiomatization and type-safety. In *International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, Aalborg, Denmark, 1998. Springer-Verlag.
32. V. F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3):292–325, 1986.
33. T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
34. T. L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
35. T. L. Veldhuizen. Arrays in Blitz++. In *ISCOPE'98*, volume 1505 of *Lecture Notes in Computer Science*, 1998.
36. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
37. E. Willink and V. Muchnick. Weaving a way past the C++ One Definition Rule. Position Paper for the Aspect Oriented Programming Workshop at ECOOP 99, Lisbon, June 14, 1999. Available at `http://www.ee.surrey.ac.uk/Research/CSRG/fog/AopEcoop99.pdf`.