An Environment for Building Customizable Software Components

Anne-Françoise Le Meur, Charles Consel, and Benoît Escrig

INRIA/LaBRI, ENSEIRB,
1 avenue du Docteur Albert Schweitzer, 33402 Talence Cedex, France,
{lemeur,consel,escrig}@labri.fr, http://compose.labri.fr

Abstract. Customization often corresponds to a simple *functional customization*, restricting the functionalities of a component to some configuration values, without performing any code optimization. However, when resources are limited, as in embedded systems, customization needs to be pushed to *code customization.* This form of customization usually requires one to program low-level and intricate transformations.
This paper proposes a declarative approach to expressing customization properties of components. The declarations enable the developer to focus on *what* to customize in a component, as opposed to *how* to customize it. Customization transformations are automatically determined by compiling both the declarations and the component code; this process produces a *customizable component.* Such a component is then ready to be custom-fitted to any application.
Besides the declaration compiler, we have developed a graphical environment both to assist the component developer in the creation of a customizable component, and to enable a component user to tailor a component to a given application.

1 Introduction

Re-usability of software components is a key concern of most software architectures. An important dimension in re-usability is the ability to customize a component for a given context. In fact, some software architectures offer a mechanism to deal specifically with customization.

JavaBeans [15] provide such a mechanism. When developing a software component (*i.e.,* a bean), a programmer can explicitly declare some variables as being the parameters (sometimes named properties) of the component customization. The component's customization parameters represent a precise interface that enables it to be tailored to a target application. Unfortunately, JavaBeans is limited to *functional customization*, *i.e.,* it restricts the behavior of a component but does not perform any *code customization.* Code customization aims to exploit the customization values to produce a smaller and faster program by reducing its generality. Such a process is particularly needed in an area such as embedded systems.

Traditionally, the scope of code customization is limited to program fragments, not software components. Code customization is commonly programmed by low-level directives, such as `#ifdef` in the C language [6]. Programs, sprinkled with directives, are transformed by a preprocessor prior to compilation. The eCos configurable OS is a recent, large scale illustration of this approach [5]. Another well-known approach to code customization is the template mechanism of the C++ language. C++ templates were originally designed to support generic programming. But it was realized that this mechanism could also express complex computations to be performed at compile time. This capability has been used intensively to design efficient scientific libraries [14,18].

Relying on the programmer to code *how* to customize a program, as proposed by current approaches, poses a number of negative consequences for a software system: (i) The development process is more error prone because code customization requires the programmer to intertwine two levels of computation – normal computation and code generation; (ii) Testing and debugging are more difficult because tools rarely deal properly with programs that generate programs; (iii) Programs are less readable and more difficult to maintain because they are cluttered with directives.

Also, customizability is limited by the extent to which the programmer is willing to explicitly encode the necessary transformations. For example, in the C++ template approach, not only does the programmer have to implement the generic program but she also has to introduce all the needed versions that can be partially computed at compile time. This situation is illustrated the appendix.

In this paper, we describe the development and deployment of customizable components. Our approach is based on a declarative language to specify *what* can be customized in a component as opposed to *how* to customize it. This approach consists of three main steps shown in Figure 1. In step 1, the programmer groups customization declarations in a *module*, on the side, as the component code is being developed. The declarations define the customization parameters of a component. In step 2, once the components and their associated declaration modules are developed, they are compiled to create a customizable component. This compilation process carries out several analyses to determine *how* to customize the component and to verify that the customization process will be performed as declared. In step 3, a transformation engine takes as input both the customizable component and the user-provided customization values and automatically generates the customized component. Ultimately, the customized component is integrated in a complete application.

We have designed a declaration language in the context of the C language and implemented the corresponding compiler. We have also developed a graphical environment both to assist the component programmer in the process of making components customizable and to enable the component user to create customized components.

We illustrate our approach with the development of a customizable data encoder for Forward Error Correction (FEC) [8]. FEC prevents losses and errors by transmitting redundant information during digital communications (*e.g.,* modems and wireless communications). Because transmission errors are inevitable during data transfer, FEC is a critical operation: it enables maximal data

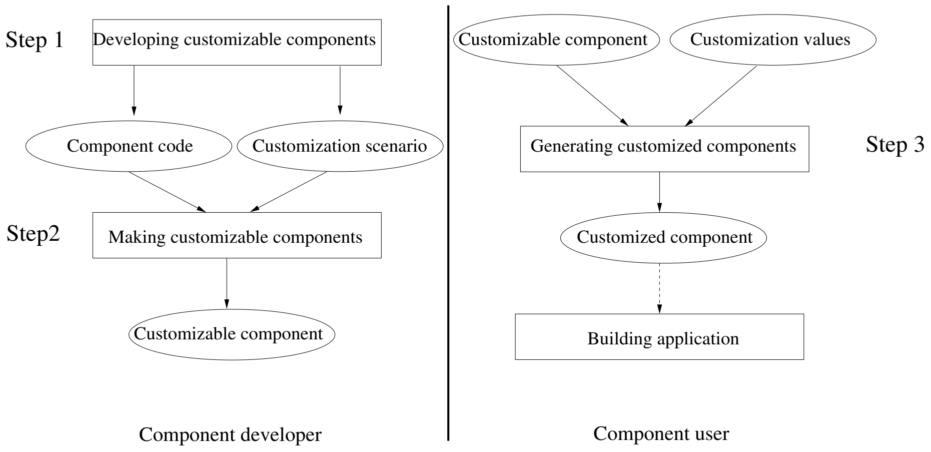


Fig. 1. Developing and deploying customizable components

transfer, using minimum bandwidth, while maintaining an acceptable quality of transmission. A FEC encoder can be customized according to four properties: the encoding method (*e.g.,* parity-bit, cyclic redundancy check), the size of the data block to be encoded, the required size of the resulting encoded data and the generator to produce the redundant data (*e.g.,* a vector, a matrix).

The rest of this paper is as follows. Section 2 details our approach and shows how our declaration language enables a developer to express the customization parameters of a component. Section 3 describes how customizable components are created. Section 4 explains the process of generating customized components. Section 5 compares the performance of customized encoders with both handwritten and generic encoders. Finally, Section 6 concludes and presents future work.

2 Developing Customizable Components

Typically, to make a component customizable, the programmer introduces parameters and some dispatching code to select appropriate behaviors and features. This parameterization also results in the definition of data structures or/and global variables.

Just as the component developer has to reason about the types of the data processed by the code, she should also reason about the customization properties of these data. That is, once the customization parameters have been determined, she should take particular care in how these parameters are used throughout the code.

The developer usually has in mind some *customization scenario* which could apply to a particular component. Indeed, when customizing a component, genericity can often be eliminated when the usage context of the component (*i.e.,* the set of the customization values) is given. To exploit this scenario, current strategies amount to programming *how* the component should be customized.

In contrast, we provide the developer with a high-level declarative language to specify *what* to customize.

2.1 Declaration Module

Following our approach, the programmer specifies customization scenarios in a *module* as a component is being developed. Because these declarations are later compiled and checked to ensure the desired customization, the programmer can write generic code without sacrificing efficiency. The customization scenario of a component defines the context in which the customization of this component is guaranteed. Since the customization aspects of a component are defined in a separate file, they do not clutter the code and can be easily modified. Furthermore, a declaration module allows an application developer to have a readable record of the customization capabilities of a component.

2.2 Intra-module Declarations

Declarations are associated with a given source program and describe the customization scenarios of specific program entities (global variables, parameterized procedures, data structures). These entities represent genericity in the component. Declarations are grouped into a module; they refer to program fragments that form a customizable software component. A program entity that is a customization parameter is said to *static*, otherwise it is *dynamic.*

Let us illustrate our approach with a component of our FEC software system. We consider the `multmat` component which multiplies a vector by a matrix. Intuitively, the vector corresponds to the data to encode, and the matrix to the generator of the redundant data. This matrix (both size and values) is a natural customization parameter because it can be noted that the matrix is fixed for a given encoder. The result of the multiplication should be stored in another vector. Pushing further the analysis, one may observe that more flexibility can be obtain by introducing an index to define where the calculated data are to be inserted in the result vector. Although, this index is not a parameter of the FEC component, it is a good candidate to be a customization parameter of the `multmat` component. The value of this parameter can be set by the callers of this component. Given these observations, the `multmat` component is implemented as follows.

```
void
multMat(int *v_in, int k, int n, int **matrix, int *v_out, int v_out_ind)
{
  int tmp, i, j;
  for(i = 0; i < n; i++)
    {
      tmp = 0;
      for(j = 0; j < k; j++)
         tmp ^= v_in[j] & matrix[j][i];
      v_out[v_out_ind + i] = tmp;
    }
}
```

Based on our analysis, a customization scenario thus declares the following parameters as static: the size of both the vector and the matrix, the matrix elements and the index in the output vector. Customizing `multMat` for some arbitrarily chosen values produces the following procedure.

```
void
multMat(int *v_in, int *v_out)
{
    v_out[0] = 0 ^ (v_in[0] & 1) ^ (v_in[1] & 0);
    v_out[1] = 0 ^ (v_in[0] & 0) ^ (v_in[1] & 1);
    v_out[2] = 0 ^ (v_in[0] & 0) ^ (v_in[1] & 1);
}
```

To achieve this kind of customization, the component developer writes the following module.

```
Module multmat {
   Defines {
     From multmat.c {
        BtmultMat :: intern multMat (D(int[]) v_in, S(int) k, S(int) n,
                                     S(int[][]) matrix, D(int[]) v_out,
                                     S(int) v_out_ind);
     }
   }
   Exports {BtmultMat;}
}
```

This declaration module, named `multmat`, is associated with the source file `multmat.c`. It defines a customization scenario, `BtmultMat`, which states that the procedure `multMat` can be customized if all its arguments but `v_in` and `v_out` are static (*i.e.,* noted `S`). This declaration includes the keyword `intern` which indicates that the source code of the procedure is available for transformation. Alternatively, a procedure can be `extern` if it is to be handled as a primitive, that is, it can be invoked when all its arguments are static, otherwise the call needs to be reconstructed.

2.3 Inter-module Declarations

Like any module system, our language for declaration modules provides *import* and *export* mechanisms. When writing a module, scenarios from another module may be imported. Similarly, exporting scenarios make them accessible to other modules.

Let us illustrate these mechanisms with the description of a component implementing a linear block coding (LBC) encoder with a systematic matrix. This component provides a procedure `systLBC` that takes two inputs, a vector and a matrix, and computes the encoded vector. This computation simply consists of both copying its input vector in the result vector and adding at the end of the vector the redundant data obtained through the multiplication of the input vector and the matrix. Intuitively, the customization scenario for the procedure `systLBC` is very similar to the one declared for the procedure `multMat`, that is, the matrix is a customization parameter. Furthermore, one can notice that if the procedure `multMat` is called to compute the redundant data, this context satisfies the `BtmultMat` scenario. Thus, the implementation of the procedure `systLBC` is

```
#include "multmat.h"
void
systLBC(int *v_in, int k, int n, int **matrix, int *v_out)
 {
  int i;
  for(i = 0; i < k; i++)
    v_out[i] = v_int[i];
  multMat(v_int, k, n - k, matrix, v_out, k);
 }
```

and the declaration module for LBC encoding with a systematic matrix is defined as follows.

```
Module lin_b_code_sys {
  Imports {
     From multmat.mdl {BtmultMat;}}
  Defines {
     From lin_b_code_sys.c {
       BtsystLBC :: intern systLBC(D(int[]) v_in, S(int) k, S(int) n,
                                   S(int[][]) matrix, D(int[]) v_out)
             {needs{BtmultMat;}};
     }
  }
  Exports {BtsystLBC;}
}
```

The keyword `needs` indicates that the invocation of `multMat` in `systLBC` satisfies the imported scenario `BtmultMat` declared in the module `multmat.mdl`. In fact, a scenario plays the role of a *customization contract* which refers to a precisely defined customization behavior. The customization context requirements specified by this contract are enforced by our analysis phase.

Once all the subcomponents of the encoder component and their associated declaration modules have been developed, they must be compiled to create a customizable component.

3 Making Customizable Components

Like any compiler, the module compiler processes the declarations to detect syntactic or semantic errors. It then verifies that the declarations are coherent with the C source code of the component. For example, it checks that customization scenarios match the type signatures of the associated C entities. The next step is the actual analysis of the component source code. Our analysis phase is automatically configured using information extracted during the module compilation, allowing the appropriate analyses to be carried out. The different analyses determine *how* to customize the component. Meanwhile, verifications are made to guarantee that this customization process will be coherent with the declared customization scenarios. A customization mismatch can either be intra-module or inter-module. In either case, it is caused by a program entity that is used in the wrong context. For example, a global variable that is declared static might incorrectly be assigned a dynamic value. In fact, solving a customization mismatch is similar to handling type problems. Our analysis phase provides the programmer with various aids to track these errors.

To assist the component programmer further, we have built a graphical environment, shown in Figure 2. It consists of a customizable component browser (bottom right corner), a component dependency visualizer (bottom left corner), and a customization scenario browser (top). The developer may use the component browser to access existing modules. When a module is selected, its cus-

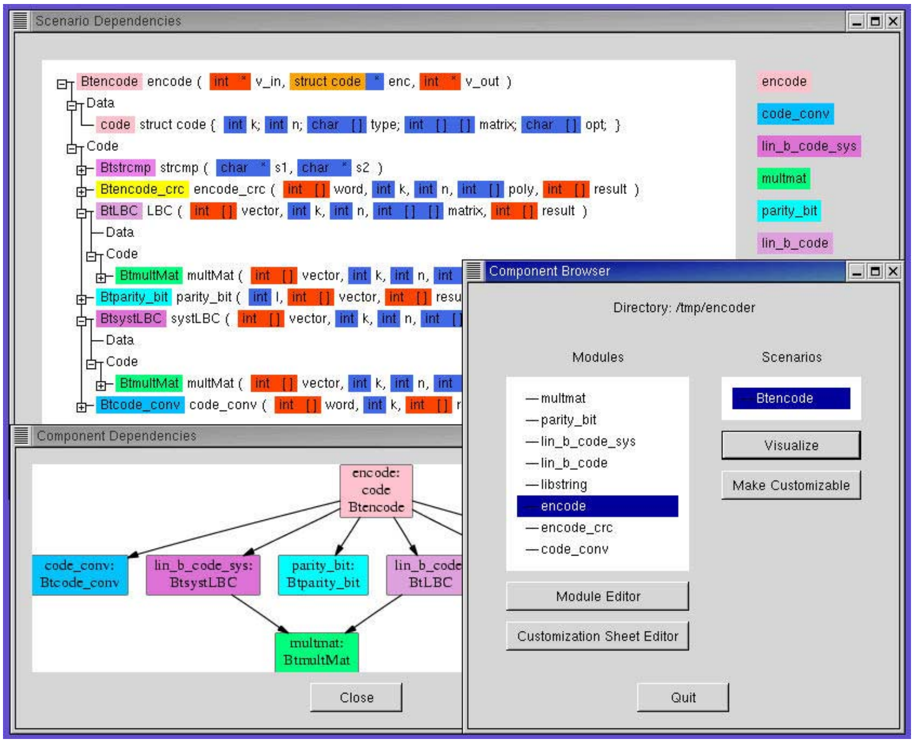
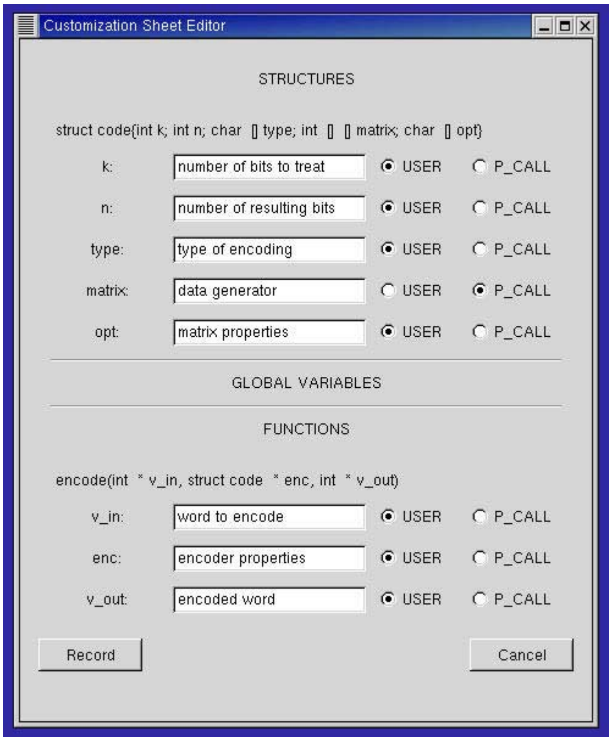

```
Btencode encode ( int * v_in, struct code * enc, int * v_out )
  Data
    code struct code { int k; int n; char [] type; int [] [] matrix; char [] opt; }
  Code
    Btstrcmp strcmp ( char * s1, char * s2 )
    Btencode_crc encode_crc ( int [] word, int k, int n, int [] poly, int [] result )
    BtLBC LBC ( int [] vector, int k, int n, int [] [] matrix, int [] result )
      Data
      Code
        BtmultMat multMat ( int [] vector, int k, int n, int
    Btparity_bit parity_bit ( int l, int [] vector, int [] resu
    BtsystLBC systLBC ( int [] vector, int k, int n, int [
      Data
      Code
        BtmultMat multMat ( int [] vector, int k, int n, int
    Btcode_conv code_conv ( int [] word, int k, int [] r
```

Fig. 2. Component developer interface

tomization scenarios are listed. For a given scenario, the corresponding hierarchy of components is displayed by the component dependency visualizer. This tool allows the developer to rapidly verify whether the component dependencies are as expected. Finer grain information are provided by the customization scenario browser: it shows the tree of sub-scenarios corresponding to a given scenario. The dependencies of a scenario consist of two parts: data and code. The data part groups the scenarios associated with data structures and global variables. The code part lists the required scenarios that are associated with procedures. Starting from the selected component scenario, the developer can recursively visit all the scenarios that are involved. Color-coded information help the developer to see how the customization parameters flow through all the scenarios. Once the developer is satisfied with the declarations, she can make the component customizable for the selected scenario. This action triggers the analysis phase.

Finally to package the customizable component, the programmer uses a customization sheet editor (see Figure 3) through which she gives a high-level description of the customization properties and defines how the user should enter the customization values: they might be typed in, or obtained through a procedure call when dealing with large values. Once the component has been made



STRUCTURES

struct code{int k; int n; char [] type; int [] [] matrix; char [] opt}

k:	number of bits to treat	[✓ USER] P_CALL
n:	number of resulting bits	[✓ USER] P_CALL
type:	type of encoding	[✓ USER] P_CALL
matrix:	data generator	USER [✓ P_CALL]
opt:	matrix properties	[✓ USER] P_CALL

GLOBAL VARIABLES

FUNCTIONS

encode(int * v_in, struct code * enc, int * v_out)

v_in:	word to encode	[✓ USER] P_CALL
enc:	encoder properties	[✓ USER] P_CALL
v_out:	encoded word	[✓ USER] P_CALL

Fig. 3. Component editor

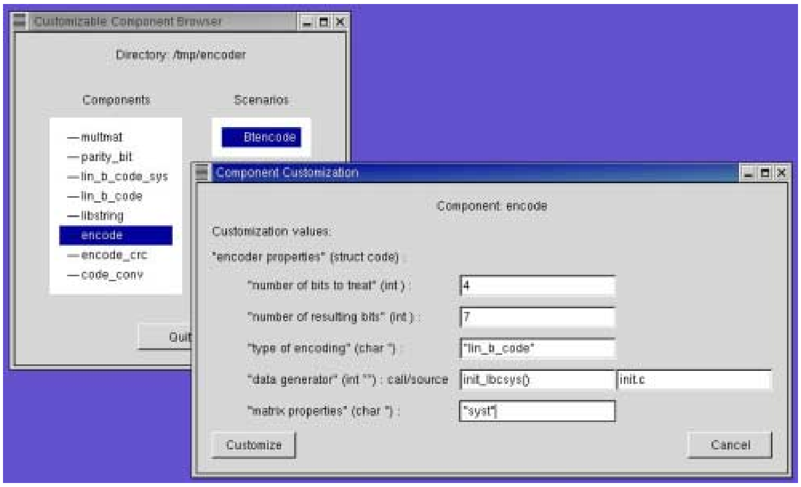
customizable, it can be distributed to component users to be custom-fitted for a particular application.

4 Generating Customized Components

Generating a customized component is performed by a transformation engine that takes two inputs: the customizable component and the customization values. As shown in Figure 4, the user may enter these values by filling in the customization sheet generated by the component developer. The customization process may be repeated as many times as there are customization values, creating each time a specific customized component. Once customized, the component is ready to be integrated in an application.

The transformation engine we use is a program specializer named Tempo [2,3]. Program specialization is an automatic technique that propagates information about a program's inputs throughout the program and performs the computations which rely on the available inputs. In our context, the propagated information is the component's customization parameters. Program specializers have been implemented for languages such as C [1,3] and Java [13], and have been successfully used for a large variety of realistic applications in domains such as operating systems [10,12], scientific algorithms [7], graphics programs [11] and software engineering [9,16,17].

However using a specializer is very complicated if one is not an expert in the domain. Furthermore, specialization was until now only applied to manually-isolated code fragments and offered no support for component specialization. Our approach enables component programmers to benefit from the power of a specializer engine without having to deal with its intricacies.



Directory: /tmp/encoder

- multmat
- parity_bit
- lin_b_code_sys
- lin_b_code
- libstring
- encode
- encode_crc
- code_conv

Btencode

Component: encode

Customization values:

"encoder properties" (struct code) :

"number of bits to treat" (int) :	4	
"number of resulting bits" (int) :	7	
"type of encoding" (char *) :	"lin_b_code"	
"data generator" (int **) : call/source	init_lbcsys()	init.c
"matrix properties" (char *) :	"syst"	

Fig. 4. Component user interface

5 Experimental Study

We have developed a customizable FEC component that covers several encoding algorithms: parity-bit (PB), cyclic redundancy check (CRC), convolutional coding (CONV), and linear block coding (LBC) for both systematic and non-systematic matrices. To assess the performance of the customized encoders, we have compared their execution time with both the execution time of manually written encoders (*e.g.,* hand-customized for a dedicated encoder configuration) and the execution time of the generic encoder.

In practice, FEC encoders are used on data of widely varying sizes, depending on the application. Thus, we have tested our encoders on very different sizes of data blocks, corresponding to real situations. For each encoder, we have measured the time spent to encode one thousand data segments. Our measurements, presented in Figure 5, were obtained using a Sun UltraSPARC 1 with 128 megabytes of main memory and 16 kilobytes of instruction and data cache, running SunOS version 5.7.

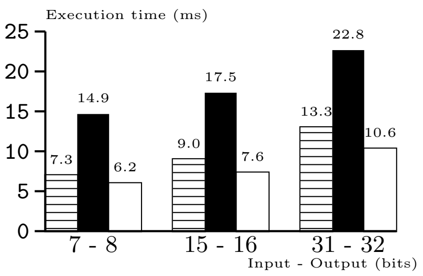
The numbers show that the encoders generated through customization are at least as efficient as the encoders manually written. The customized LBC encoder is even 4 times faster than the manually written LBC encoder (see graph 5-d). This speedup is mainly due to code optimization, like loop unrolling and constant folding, performed by our customization process but that are tedious to perform by hand. As expected, the hand-written encoders are more efficient than the generic one. However, the difference is negligible in the CRC case shown in the graph 5-f. This is due to the fact that the inefficiencies in the generic encoder are canceled by the cost of the other calculations that are performed.

We mentioned above that loop unrolling operations were performed. However, it may happen that such transformation is not desired when the number of loop iterations is high. To address such a situation, our declaration language provides a means to easily express constraints on the property value to propagate, and thus to avoid loop unrolling if needed.

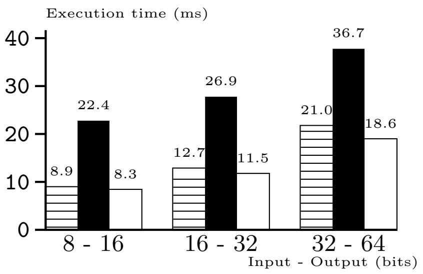
6 Conclusion and Future Work

We have presented an approach that provides developers with a language and an environment to create customizable components. They can then be customized for specific applications. Our approach relies on a declaration language that enables the component programmer to specify the customization properties of a software component. The declarations consist of a collection of customization scenarios that are associated with the program entities. The scenarios of a component do not clutter the code; they are defined aside in a module.

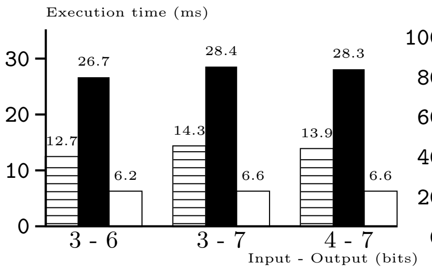
Customizable components are created from the declarations and the component code through a compilation process. This phase mainly corresponds to carrying out several analyses to automatically determine how to customize the component accordingly to the declared customization values. Once generated, customizable components can be distributed to be tailored by application



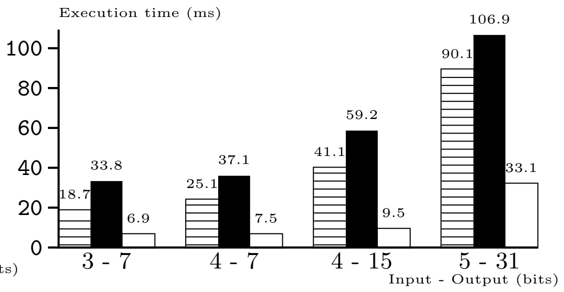
5-a) Parity-bit



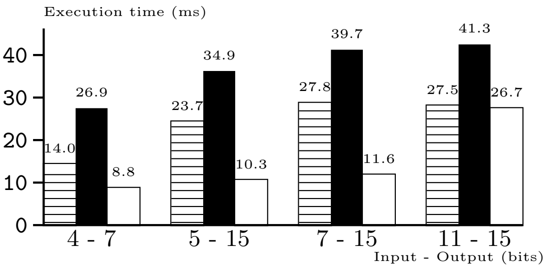
5-b) Convolutional coding



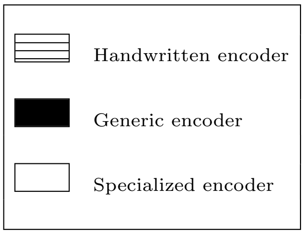
5-c) Linear block coding with systematic matrix

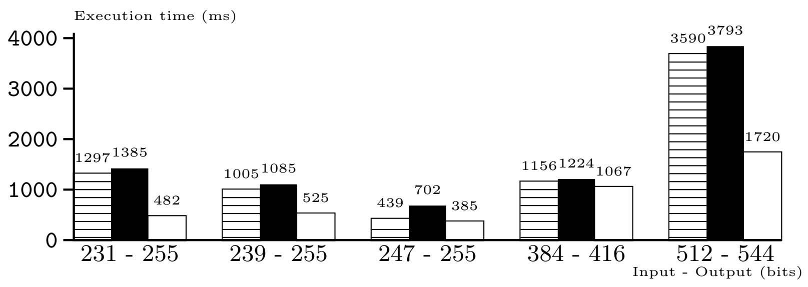


5-d) Linear block coding



5-e) Cyclic redundancy check





5-f) Cyclic redundancy check

Fig. 5. FEC encoder benchmarks

builders. The component customization values are set through a customization sheet. Besides the language compiler, we have developed a set of tools to assist the developer at each step of the process.

We have applied our approach to create a customizable forward error correction encoder. The customized components generated for various usage contexts have exhibited performance comparable to, or better than manually customized code.

We are now working on a larger scale experiment where our technology is applied to build audio applications from customizable components such as filters, buffers, *etc.* In the near future, we also plan to extend our approach to provide dynamic re-customization capabilities.

Acknowledgments

This work was partially supported by the ITEA project ESAPS.

References

1. L.O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.
2. C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), 1998.
3. C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.
4. K. Czarnecki and U. W. Eisenecker. *Generative Programming - Methods, Tools, and Applications.* Addison-Wesley, 2000.
5. Red Hat. eCos : Embedded configurable operating system, 2000. http://sources.redhat.com/ecos.
6. B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
7. J.L. Lawall. Faster Fourier transforms via automatic program specialization. In J. Hatcliff, T.Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, volume 1706 of *Lecture Notes in Computer Science*, pages 338–355, Copenhagen, Denmark, 1999. Springer-Verlag.
8. S. Lin and D. J. Costello. *Error Control Coding: Fundamentals and Applications.* Prentice Hall: Englewood Cliffs, NJ, 1983.
9. R. Marlet, S. Thibault, and C. Consel. Efficient implementations of software architectures via partial evaluation. *Journal of Automated Software Engineering*, 6(4):411–440, October 1999.
10. G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

11. F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, Chicago, IL, May 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.
12. C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.
13. U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 367–390, Lisbon, Portugal, June 1999.
14. J. G. Siek and A. Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.
15. Java Sun. Javabeans component architecture. http://java.sun.com/products/javabeans/.
16. S. Thibault and C. Consel. A framework for application generator design. In *Proceedings of the Symposium on Software Reusability*, Boston, MA, USA, May 1997.
17. Scott Thibault, Renaud Marlet, and Charles Consel. A domain-specific language for video device driver: from design to implementation. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.
18. T.L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments*, number 1505 in Lecture Notes in Computer Science, Santa Fe, NM, USA, December 1998. Springer-Verlag.

Appendix

Let us consider the function `power` which raises `base` to the power of `expon`. The usual C code of this function looks like:

```
int power(int base, int expon)
{
  int accum = 1;
  while (expon > 0) {
    accum *= base;
    expon--;
  }
  return(accum);
}
```

Now let us suppose that the value of `expon` is known at compilation time. If the value of `expon` is not too large, it is thus interesting to optimize the code by unrolling the loop that depends on it.

Using our approach, the programmer writes the following customization scenario:

```
Btpower :: power(D(int) base, D(int) expon)
          { constraint{ expon : expon < 10; } };
```

which enables the loop to be unrolled if the value of `expon` is less than 10.

Thus, once customized, say for a value of `expon` equals to 3, the code of the `power` function is:

```
int power(int base)
{
  return base*base*base;
}
```

In C++, if the programmer decides that it is worth to unroll the loop in some cases, she must write, besides the generic code (which looks very much like the C version), yet another implementation of the `power` function. This other implementation explicitly tells the compiler to perform the desired transformations. In this situation, the "trick" is to use the compiler to recursively inline the function `power`. Here is how to do it:

```
template<int expon>
inline int power(const int& base)
{ return power<expon-1>(base) * base; }

template<>
inline int power<1>(((const int& base)
{ return base; }

template<>
inline int power<0>(const int& base)
{ return 1;}
```

This way, the call `power<3>(base)` is successively transformed by the compiler as follows: `power<3>(base)`, `power<2>(base) * base`, `power<1>(base) * base * base`, and finally `base * base * base`.

There exist numerous examples of such situations where the programmer has to write the code to enable the compiler to perform the needed transformations [4].

In our approach, the programmer has just to declare *what* to optimize and does not have to worry about *how* to implement the optimization.