Reuse: Right Idea, Wrong Representation?

Ted J. Biggerstaff
Software Generators, LLC
Austin, Texas USA
dslgen at softwaregenerators dot com

Abstract—**This paper introduces the DSLGen™ (Domain Specific Language Generator), an execution platform independent program generation system that produces optimized code (without reprogramming) for multiple execution platform architectures. The target computation is specified in a programming language independent, Implementation Neutral Specification (INS) form and need not ever be changed unless the target computation changes. The target execution platform architecture is specified separately in terms of high level domain specific descriptors. Only the platform specification needs to change when moving to a new platform. Key to this solution is the introduction of new representational abstractions for the early stages of generation. The overall conclusion from this research is that the new representational abstractions significantly amplify the reusability of the underlying componentry.**

Keywords -- associative programming constraints, natural and synthetic partitions, design patterns, logical and physical architectures, implementation neutral specification, domain specific languages, inference, problem domain inference, partial evaluation.

I. INTRODUCTION

A long standing problem in reuse has been amplifying the reusability of components so that a small set of components can be composed to form many programs. In many cases, two components that conceptually should be compatible are often not for relatively trivial reasons. For example, composing two components that perform two sequential transformations on data may execute too slowly because of underlying computational redundancies or because sequential operations have been imposed on the operations where some degree of parallelism would make them sufficiently fast.

Many techniques have been tried to solve this problem with varying degrees of success but none can be said to really solve the amplification problem in any general or sufficient sense.

II. THE UNDERLYING PROBLEM

After struggling with this problem for a long time, I have concluded that the key difficulty lies in the representation used to express reusable components. Virtually all component representations are either based on programming languages (PLs) or based on abstractions thereof. This, I believe, is the underlying problem. PL representations (at any level of abstraction) are inherently highly detailed, highly restrictive and overly constraining. PLs require too many design decisions to be made too early just to write a segment of code (or even to express code based abstractions). Since those design decisions often constrain the remainder of the program broadly, early and subtly, they often impose many invisible restrictions on the contexts in which that code will work. For example, thread packages often provide a framework for user written thread routines but only allow a single user parameter to be sent to the user's thread routine. This requirement permeates broadly beyond the user's thread routine and may require broad modifications. This means the user must add "plumbing" code (e.g., via global variables or packaged parameters) to provide the data that is needed by the thread routine. This kind of restriction is highly problematic for reuse in that the programmer must have knowledge of the internals of a reusable component (i.e., "white box" reuse) whereas the ideal would be that he should need to have no knowledge of the internals of a component (i.e., "black box" reuse). This is especially problematic for generation systems that seek to automatically use or generate such components. Previously, generating "plumbing" code in such situations has been beyond the state of the art.

III. THE SOLUTION

To address the problems induced by PL-based representations, DSLGen™ introduces a new kind of abstraction, the "Associative Programming Constraint" (APC). APCs modify computational specifications somewhat like adjectives and adverbs modify other grammatical elements in natural language. APCs thereby **imply** some properties of the programming structures that will eventually be expressed in the target computational but do so without yet explicitly formulating those programming structures. They defer the generation of PL-based elements until the full architecture of the target program has been worked out. For example, an APC might express the nominal form of a loop implied by some domain operator (e.g., a convolution operator) or by a domain specific operand (e.g., a grayscale image expressed as an array of pixels). Alternatively, another kind of APC might describe an "early draft" partitioning of a computation without yet actually performing that partitioning.

APCs are partial and provisional. They are partial in the sense that they specify only a singular design feature of a programming component without fully defining the structural details of that component or its context. They can be composed to fill out the full set of design features of that component.

It is as important to understand what APCs do not say as what they do say. For example, a loop APC does not provide sufficient information by itself to write the code for a loop. It

does not specify how that loop is partitioned, or whether that loop executes within a parallel thread, or whether that loop executes partly within threads and partly outside of threads, and so forth. It provides no knowledge of the exact context in which that loop will be written. Nor does it determine whether the loop occurs within a function and if so, which one. These elements remain to be specified by other elements of the design structures (e.g., other APCs).

Furthermore, APCs are provisional. That is, they may (and almost certainly will) be changed as the generation process progresses. They may be combined. They may be formed into sets. Those sets may be combined. For example, there is a kind of algebra for APCs that provides the rules of formulation, transition, combination and reorganization.

More broadly speaking, APCs are composed to form Logical Architectures (LAs) that modify computational specifications. Those LAs evolve into forms that will guide the generation of cloned, specialized versions of the computation specification that they modify.

Furthermore, the LAs will determine a Design Framework, (i.e., the global architecture for the PL based computation) into which cloned and specialized computational specifications will be installed, where the specialization will be determined in part by the LAs.

IV. AN EXAMPLE PROBLEM

The initial problem domain treated by DSLGen™ is digital signal processing (DSP) and includes problems that range from signal and image processing to neural networks to pattern recognition along with a rich set of related problems. The domain specific language used to express the Implementation Neutral Specification (INS) of a computation is based on the Image Algebra (IA) [9]. The INS will never have to be reprogrammed regardless of the evolution of and changes to the execution platform.

As an example computation, we develop a program that performs Sobel edge detection on a grayscale image (i.e., where the pixels are shades of gray). Such a program would take, for example, the image "a" in Fig. 1 as input and produce the image "b" in Fig.2 as output. The output image has been processed so as to enhance (line) edges of items in the image by the Sobel edge detection method.

Each black and white pixel b[i,j] in the output image "b" is computed from an expression involving the sum of products of pixels in a neighborhood (e.g., defined by "sp", of type *iatemplate)* surrounding the a[i,j] pixel and the coefficients defined by that neighborhood (e.g., sp). This is called a *convolution* of a matrix with a template (or neighborhood). In the IA, a convolution is designated by the $\oplus$ operator, e.g., (a $\oplus$ sp). In the following examples, s and sp will designate instances of the class iatemplate. Mathematically, the Sobel computation is defined as

$$\{\text{Forall}_{i,j}\ (b_{i,j} : b_{i,j} = \text{sqrt}((\textstyle\sum_{p,q} (w(s)_{p,q} * a_{i+p,j+q})^2 + \sum_{p,q} (w(sp)_{p,q} * a_{i+p,j+q})^2)\} \quad (1)$$

where i and j are indexes that range over the matrices a and b; p and q are indexes that range over the iatemplate neighborhoods s and sp; and the coefficients of each neighborhood (which are also called *weights*) are defined by the function "w". For Sobel edge detection, the weights are all defined to be 0 if the center pixel of the neighborhood corresponds to an edge pixel in the image (i.e., w(s) = 0 and w(sp) = 0), and if not an edge pixel, they are defined by the s and sp neighborhoods shown in (2). It is convenient to index the neighborhoods in the DSL from -1 to +1 for both dimensions so that the current pixel being processed is at (0, 0) of the neighborhood.

$$w(s) = \begin{matrix} \\ P \\ \\ \end{matrix} \overbrace{\begin{matrix} -1 & 0 & 1 \end{matrix}}^{Q} \begin{matrix} -1 \\ 0 \\ 1 \end{matrix} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad w(sp) = \begin{matrix} \\ P \\ \\ \end{matrix} \overbrace{\begin{matrix} -1 & 0 & 1 \end{matrix}}^{Q} \begin{matrix} -1 \\ 0 \\ 1 \end{matrix} \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad (2)$$

Since an implementation of this computation for a parallel computer may not be organized like the mathematical formula, it is useful to represent this specification more abstractly because such abstractions can defer the implementation and organization decisions and thereby allow the computation (i.e., "what" is to be computed) to be specified completely separately and somewhat independently from the implementation form (i.e., "how" it is to be computed). Thus, the abstract computation specification (i.e., the INS) is independent of the architecture of the machine that will eventually be chosen to run the code. Choosing a different machine architecture for the implementation form without making any changes to the specification of the computation (i.e., to the "what"), will automatically generate a different implementation form that is tailored to the new machine's architecture. More to the point, porting from one kind of machine architecture (e.g., machines with instruction level parallelism like Intel's SSE instructions) to a different kind of machine architecture (e.g., machines with large grain parallelism such as multi-core CPUs) can be done automatically by only making trivial changes to the machine specifications and no changes to the computation specification (i.e., to the "what"). The publication form in [9] for the Sobel Edge detection mathematical formula (1) is based on the Image Algebra domain specific language (DSL). Re-expressing the formula (1) in the Image Algebra gives a first cut at the INS for the Sobel example:

$$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2} \quad (3a)$$

Of course, the INS will need some declarations for a, b, s, sp, etc.:

```
(DSDeclare IATemplate s :form (array (-1 1) (-1 1))
        :of DSNumber)
(DSDeclare IATemplate sp :form (array (-1 1) (-1 1))
        :of DSNumber)
(DSDeclare DSNumber m :facts ((> m 1)))
(DSDeclare DSNumber n :facts ((> n 1)))
(DSDeclare BWImage a :form (array m n) :of BWPixel)
(DSDeclare BWImage b :form (array m n) :of BWPixel)
```

$$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2} \quad (3b)$$

m and n are assumed to be user defined elsewhere. The DSL type declarations (e.g., IATemplate, BWImage, etc.) define CLOS types that will eventually refine to C types. The ":facts" keyword denotes a conjunction (i.e., list) of facts pertinent to the declared item (e.g., m) and will be used to infer, for example, that"(i==(m-1))" is false when "(i==0)" is true. Beyond (3b), we will also need some definitions for ⊕ and for s and sp, which must be equivalent to (2). All of these definitions will be provided later.

The IA DSL is the basis of the Implementation Neutral Specification (INS) in the examples used throughout the remainder of this document. A full description of the IA used by DSLGen™ is beyond the scope of this paper (see [9]) but a few comments are in order. The IA is much like APL in the sense that IA specifications eschew the use of explicit looping constructs allowing loops to be implied by IA operators and data structures. The generator will introduce implied loops as constraints and, through the manipulation, combination and propagation of these constraints, will determine the relationships between IA expressions and loops. The initial form of the LA arises during this process.

Figure 1. Input Image a

In DSLGen™, the Image Algebra is adapted to a more utilitarian, LISP based syntax with prefix operators, without the pretty symbols (e.g., the convolution operator ⊕ becomes a Lisp symbol), and with the w functions in (1) becoming so-called *Method-Transforms* (*MT*), which rewrite *Abstract Syntax Tree* (*AST*) subtrees. MTs look superficially a bit like object oriented methods with a pattern (i.e., the MT's *left hand side* or *lhs*) as the analog of a method's parameter sequence and a pure functional expression *right hand side* (*rhs*) as the analog of a method's body. MTs will be an important component of the *intermediate language* (*IL*) by which provisional but malleable definitions are expressed. For example, w of the neighborhood s is an MT expressed as:

```
(Defcomponent w (sp  #. ArrayReference ?p ?q)
  (if (or (== ?i  ?ilow) (== ?j  ?jlow)
          (== ?i ?ihigh) (== ?j ?jhigh)
            (tags (constraints partitionmatrixtest edge)))
      (then 0)
      (else (if (and (!= ?p 0) (!= ?q 0))
                (then ?q)
                (else (if (and (== ?p 0) (!= ?q 0))
                          (then (* 2 ?q))
                          (else 0)))))))                    (4)
```

where ArrayReference is the name of a shared pattern that will recognize an array reference in an AST (e.g., a[i,j]) and bind the loop index variables (e.g., i and j) to the pattern variables ?i and ?j, the matrix name a to ?a and the expressions defining the upper and lower ranges of those loop indexes to ?ihigh, ?ilow, etc. The remainder of the lhs pattern after ArrayReference will bind ?p and ?q to the loop index names used by the inner convolution loops over the neighborhood designated by sp. The w definition of neighborhood s follows a similar pattern.

The "tags" expression designates a property list for the OR conditional expression, which in (4) provides the user supplied domain knowledge that the OR expression is a *partitioning condition* for this computation that will identify *edge* partitions and by implication, a non-edge (i.e., *center*) partition. Because such partitions are specific to particular computation specifications, they are designated by the moniker "*natural"* partitions. Later this notion will be extended to add a new kind of partition, the "*synthetic partition,*" which will provide a mechanism for incorporating implementation requirements into the LA.

Problem domain concepts like "edge" and "center" play a key role in the logical architecture for the target computation and beyond that, in imposing design pattern frameworks onto a logical architecture. Heuristic rules based on domain concepts are the mechanisms whereby DSLGen™ chooses a design pattern framework to introduce PL structures and clichés (e.g., coordinated routines, synchronization patterns and thread management clichés). It then maps the LA into the structures and clichés of that design pattern framework.

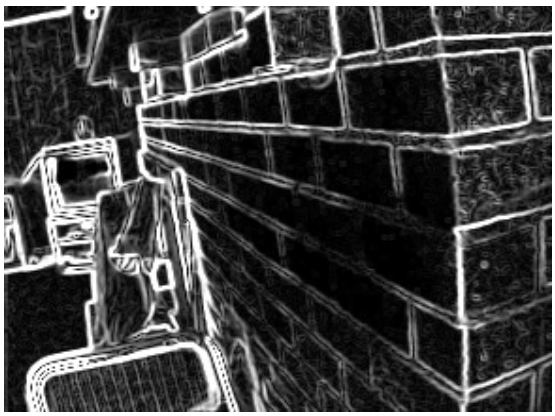

Figure 2. Output Image b

The opportunity for such domain specific heuristic rules is open ended, especially given the rich variety of possible semantic subclasses of partitions. Different problem

examples may introduce other domain semantics. For example, in the matrix domain, the semantic subclasses include *corners* (e.g., corners are special cases in the partitioning of image averaging computations); *non-corner edges* also used in image averaging; *upper and lower triangular* matrices, which are used in various matrix algorithms; *diagonal* matrices; and so forth. By contrast, in the data structure domain, domain subclasses include *trees*, *left and right subtrees*, *red and black nodes*, etc. In general, domain concepts drive the DSLGen™ program generation process.

V. Overview of the Generation Process

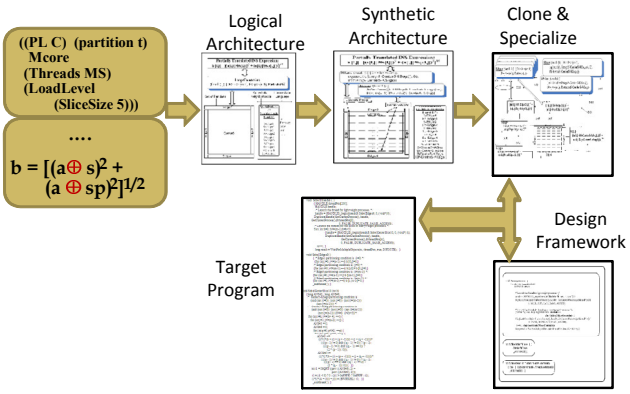

Figure 3. Generation Process

It is beyond the limitations of this paper to describe in detail how DSLGen™ generates a target program but we can sketch out the process at a high level of abstraction. Figure 3 is an overview of the process. The target machine is specified by the high level descriptor expression "((PL C) (partition t) (threads MS) (LoadLevel (sliceSize 5)))". This specifies "C" as the output language. It asks for partitioning of the computation, which will make use of the property list "(tags (constraints partitionmatrixtest edge))" from expression (4) to accomplish the partitioning. The result will be the partition set {edge1, edge2, edge3, edge4, center5} from the S neighborhood and partition set {edge6, edge7, edge8, edge9, center10} from the SP neighborhood. In the end, these two partition sets will be combined into the set {edge11, edge12, edge13, edge14, center15} where edge11 is the combination of edge1 and edg6 with the others (i.e., edge12 through center15) following a similar pattern.

Newly specialized neighborhoods will be created for each partition (e.g., s-edge11 and sp-edge11). Importantly, the definitions of w(s-<partition-M>) and w(sp-<partition-M>) will be specialized for each <partition-M> by assuming the partitioning condition of that partition is true (e.g.," (== ?i ?ilow)" is true) and partially evaluating the body of the definition. For example, the body of w(s-edge-M) and w(sp-edge-M) definitions (for all M's) will partially evaluate to 0. The neighborhood center partitions will reduce the body of expression 4 to the expression representing the false branch of the partitioning expression. When finally inlined into the generated code, the edge partitions will collapse into single loops processing an edge and setting the pixel value to 0 (i.e., white). The logic generated for center partitions will mimic the structure of the else branch of expression (4) and analogously for the s neighborhood.

In the Synthetic Architecture phase, the synthetic architecture (which derives from the LA) can evolve in many possible directions based on the value of remainder of the execution platform specification. For example, a vector machine description would produce expressions made up of vector instructions (e.g., Intel SSE instructions). Alternatively, the lack of any machine specific designations would produce a simple Von Neumann program. Finally, for the example given, "(threads MS) (LoadLevel (sliceSize 5))," the synthetic architecture will introduce new loops (e.g., one to slice the center partition into slices and a second loop to process a slice). It will also introduce a new "synthetic" partition to represent the center with slices and another new synthetic partition to represent one of the slices from the set of slices.

Synthetic partitions extend the "natural" partition concept by adding design feature requirements that will engender architectural variations beyond simple partitioning. For example, our example platform specification requires separation of the nominal loop over the image into two parts: one that determines the start and end indexes of each slice and a second one that processes a single slice. The requirement "(threads MS)" will trigger a search for a Design Framework (DF) that handles slicer/slicee synthetic architectures (i.e., like the one generated by this example) with the additional requirement that the DF introduces thread based parallelism in the overall computation. A DF is a formalization of the "Design Patterns" concept from the gang of four book of the same name [7]. DF's comprise patterns from that book as well as the book "Patterns for Parallel Programming" [8].

To prepare the partially translated INS specification for insertion into a DF that will establish the architectural scaffolding (among other services) for the target implementation structure, the generator must clone the INS and specialize it to the partitions that are specific to the "holes" in the DF. Expressions (5) and (6) respectively illustrate specializations for an example edge and the center slice. Eventual inlining of the definitions for ⊕ and the specialized Method-Transforms such as w(s-<partition-M>) along with partial evaluation will do the lion's share of the work in formulating the final code. There are other facilities that handle other specialization processes but they are beyond the scope of this paper. The broad sense of the processing is sufficiently provided by this level of detail.

$$b[i,j]=[(a[i,j] \oplus \text{S-Edge2}[p,q])^2 + (a[i,j] \oplus \text{sp-Edge2}[p,q])^2]^{1/2} \quad (5)$$

$$b[i,j]= [(a[i,j] \oplus \text{S-Center5-ASeg}[p,q])^2 + (a[i,j] \oplus \text{sp-Center5-ASeg}[p,q])^2]^{1/2} \quad (6)$$

The DF is a skeletal structure that mixes fairly low level code (e.g., for thread management and synchronization) with "holes" that are subject to a few simple, general restrictions. For example, one of the holes is designed to sequentially batch process lightweight threads (i.e., threads that may cost as much time to set up as they might save by being run in parallel). Here is where domain knowledge based heuristics come into play. One of the domain characteristics of edge partitions is that they tend to be lightweight computations. Thus, the generator makes the decision to relegate the code specific to the edge partitions to this hole based on heuristic knowledge.

Figure 4 illustrates the essence of the code that is generated for the chosen example (with some cosmetic positional editing for presentation and some added comments for improved understandability).

```
/* THREAD MANAGEMENT ROUTINE*/

void  SobelThreads8 (  )
    { HANDLE threadPtrs[200];
      HANDLE handle;
       /* Launch the thread for lightweight processes. */
      handle = (HANDLE)_beginthread(& SobelEdges9,
                                          0, (void*) 0);
     DuplicateHandle(GetCurrentProcess(),handle,
                          GetCurrentProcess(),&threadPtrs[0],
                        0,FALSE,
                        DUPLICATE_SAME_ACCESS);

     /* Launch the threads for the slices of heavyweight
        processes. */
    for ( int h=0; h<=(m-1);h=h+5)
        {handle=(HANDLE)
          _beginthread(&SobelCenterSlice10,0,(void*)  h);
          DuplicateHandle(GetCurrentProcess(), handle,
                  GetCurrentProcess(),&threadPtrs[tc],
                 0, FALSE, DUPLICATE_SAME_ACCESS);
          tc++;  }
    long result = WaitForMultipleObjects(tc, threadPtrs, true,
                                          INFINITE);   }

    /* BATCHED PROCESSING OF EDGES*/

    void SobelEdges9( )
          { /* Edge1 partitioning condition is  (i=0) */
            {for (int j=0; j<=(n-1);++j) b [0,j]=0;}
           /* Edge2 partitioning condition is  (j=0) */
           {for (int i=0; i<=(m-1);++i) b [i,0]= 0;}
           /* Edge3 partitioning condition is  (i=(m-1)) */
           {for (int j=0; j<=(n-1);++j) b [(m-1),j]=0;}
           /* Edge4 partitioning condition is  (j=(n-1)) */
           {for (int i=0; i<=(m-1);++i) b [i, (n-1)]= 0;}
           _endthread( ); }
```

```
/*THREAD ROUTINE FOR A CENTER SLICE*/

void SobelCenterSlice10 (int h)
{long ANS45;  long ANS46;
 /*  Center5-KSegs partitioning condition is
     (and (not (i=0))  (not (j=0))  (not (i=(m-1)))
          (not (j=(n-1)))) */
  /*  Center5-ASeg partitioning condition is
      (and (not (i=0))  (not (j=0))  (not (i=(m-1)))
           (not (j=(n-1))) (h<=i) (i<=(min (h+4) (m-1)))*/
  for (int i=h; i<=(min (h+4) (m-1)); ++i) {
     for (int j=1; j<=(n-2); ++j) {
        ANS45 = 0;
        ANS46 = 0;
        for (int p=0; p<=2; ++p) {
          for (int q=0; q<=2; ++q) {
     ANS45 +=
      (((*((*(b + ((i + (p + -1))))) + (j + (q + -1)))))*
         ((((p - 1) != 0) && ((q -  1) != 0)) ? (p - 1):
                 ((((p - 1) != 0) && ((q -  1) == 0)) ?
                     (2 * (p - 1)): 0)));
     ANS46 +=
      (((*((*(b + ((i + (p + -1))))) + (j + (q + -1)))))*
         ((((p - 1) != 0) && ((q -  1) != 0)) ? (q - 1):
                  ((((p - 1) == 0) && ((q -  1) != 0)) ?
                      (2 * (q -  1)): 0)));   }}
     int i1 = ISQRT ((pow ((ANS46), 2) +
                          pow ((ANS45), 2)));
      i1 = (i1 < 0) ? 0 : ((i1 > 0xFFFF) ? 0xFFFF : i1);
      ((*((*(A + (i))) + j))) = (BWPIXEL)  i1;   }}
       _endthread( ); }
```

Figure 4. Multicore Code

The ANS45 and ANS46 expressions mimic the structures in the definitions of W(sp …) shown as expression (4) and of W(s …), whose definition is not shown. By contrast, Figure 5 shows a segment of code that also computes a pixel of the center partition but for the case where the platform specification requests Instruction Level Processing (i.e., vector instructions) using Intel's SSE instruction set. This example is a color image with RGB (Red-Green-Blue) color planes and Figure 5 shows the computation of only one color plane.

```
/*EDGE COMPUTATIONS PRECEED THIS SEGMENT*/

int IDX4   = 0;   int IDX3   = 0;
int DSARRAY9  [3] [3]  = {{-1,-2,-1},{0,0,0},{1,2,1}};
int DSARRAY10  [3] [3]  = {{-1,0,1},{-2,0,2},{-1,0,1}};
for (IDX3=1; IDX3<=98; ++IDX3)
  {{for (IDX4=1; IDX4<=98; ++IDX4)
     {{/* Cloned loop specialized for design object
        (SPPART-0-CENTER10 SPART-0-CENTER5).*/
       /*Cloned loop specialized for partitioning conditions:
       (!= IDX4 99) (!= IDX3 99) (!= IDX4 0)  (!= IDX3 0)*/
        {{(ANSCOLORPIXEL4.RED1) =
            UNPACKADD (
```

```
PADD (2,
  PADD (2,
    PMADD (3,(& ((*((*(C + ((IDX3 -  1)*100)))
                    + (IDX4 -  1))).RED1)),
        (& (*((*(DSARRAY9 + (0*3))) + 0)))),
    PMADD (3,(& ((*((*(C + (IDX3*100)))
                    + (IDX4 -  1))).RED1)),
        (& (*((*(DSARRAY9 + (1*3))) + 0))))),
   PMADD (3,(& ((*((*(C + ((IDX3 +  1)*100)))
                    + (IDX4 -  1))).RED1)),
      (& (*((*(DSARRAY9 + (2*3))) + 0))))));
 (ANSCOLORPIXEL2.RED1) =
  UNPACKADD (
   PADD (2,
    PADD (2,
     PMADD (3,(& ((*((*(C + ((IDX3 -  1)*100)))
                    + (IDX4 -  1))).RED1)),
        (& (*((*(DSARRAY10 + (0*3))) + 0)))),
     PMADD (3,(& ((*((*(C + (IDX3*100)))
                    + (IDX4 -  1))).RED1)),
        (& (*((*(DSARRAY10 + (1*3))) + 0))))),
   PMADD (3,(& ((*((*(C + ((IDX3 +  1)*100)))
                    + (IDX4 -  1))).RED1)),
      (& (*((*(DSARRAY10 + (2*3))) + 0))))); }
((*((*(D + (IDX3*100))) + IDX4)).RED1) =
  ISQRT ((pow ((ANSCOLORPIXEL2.RED1),2)
        +  pow ((ANSCOLORPIXEL4.RED1),2)));

/* … REMAINING COLOR PLANES COMPUTED
     HERE */
```

Figure 5. SSE Instruction Set Code

The key point of Fig. 5 is that the organization of the code is radically different from Fig. 4. Notice that the generated arrays DSARRAY9 and DSARRAY10 contain the constants defined in expression (2). The generator constructed these array definitions by partially evaluating the definitions of w(sp ….) and w(s…) for all indexes in the neighborhoods sp and s. The PMADD functions are C macros that set up the SSE instruction PMADD, which will do a product multiply and add of one row of the weights in the generated arrays and the corresponding vector of pixel values in one color plane of the image C. The PADD macros invoke PADD instructions to add the row results together and UNPACKADD unpacks the results from the SSE register. Each of the generated answer variables ANSCOLORPIXEL2.RED1 and ANSCOLORPIXEL4.RED1 contains the partial results from one of the two separate convolution operations in expression (3a).

The overall conclusion from this work is that the new representational abstractions lead to significant amplification of the underlying reusable componentry.

VI. Related Research

A full consideration of related and previous research is beyond the scope of this paper. However, some broad differences between this work and all previous research stand out and illustrate the novelty of this work. Most generally, this generator eschews the programming language domain and representation during the early generation process. This leads to significant advantages such as the ability to work with logical architecture abstractions (e.g., APCs) and evolve the shape of the desired overall architecture before casting the computation into a programming language form. During this early "design" process, many lower level details can be ignored or elided thereby vastly simplifying the evolution of the architecture.

Because the computation is specified in an implementation neutral form, reprogramming is avoided. Only the specification of the execution platform needs to be changed when moving to a new execution platform. In theory, a new generation regime for DSLGen™ can be added to support any new execution platform regardless of the architectural structure and the code generated for that platform will exploit its unique features. Additionally, new application domains can be added by similar techniques.

References

[1] Ted J. Biggerstaff, "A perspective of generative reuse, annals of software engineering," Baltzer Science Publishers, AE Bussum, The Netherlands, 1998, pp.169-226.

[2] Ted J. Biggerstaff, "Fixing some transformation problems" Automated Software Engineering Conference, Cocoa Beach, Florida, 1999, pp. 10.

[3] Ted J. Biggerstaff, "A new architecture of transformation-based generators," IEEE Transactions on Software Engineering, Vol. 30, No. 12, Dec., 2004, 1036-1054.

[4] Ted J. Biggerstaff, "Automated partitioning of a computation for parallel or other high capability architecture," Patent no. 8,060,857, United States Patent and Trademark Office, filed January 31, 2009, issued November 15, 2011.

[5] Ted J. Biggerstaff, "Non-localized constraints for automated program generation," United States Patent and Trademark Office, Patent no. 8,225,277, filed April 25, 2010, issued July 17, 2012.

[6] Ted J. Biggerstaff, "Synthetic partitioning for imposing implementation design patterns onto logical architectures of computations," United States Patent and Trademark Office, Patent no. 8,327,321, filed August 27, 2011, issued Dec. 4, 2012.

[7] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, Addison-Wesley, 1995.

[8] Timothy G. Mattson, Beverly A. Sanders and Berna L. Massingill, Patterns for Parallel Programming, Addison Wesley, 2008.

[9] Gerhard X. Ritter and Joseph N. Wilson, The Handbook of Computer Vision Algorithms in Image Algebra, CRC Press, 1996.