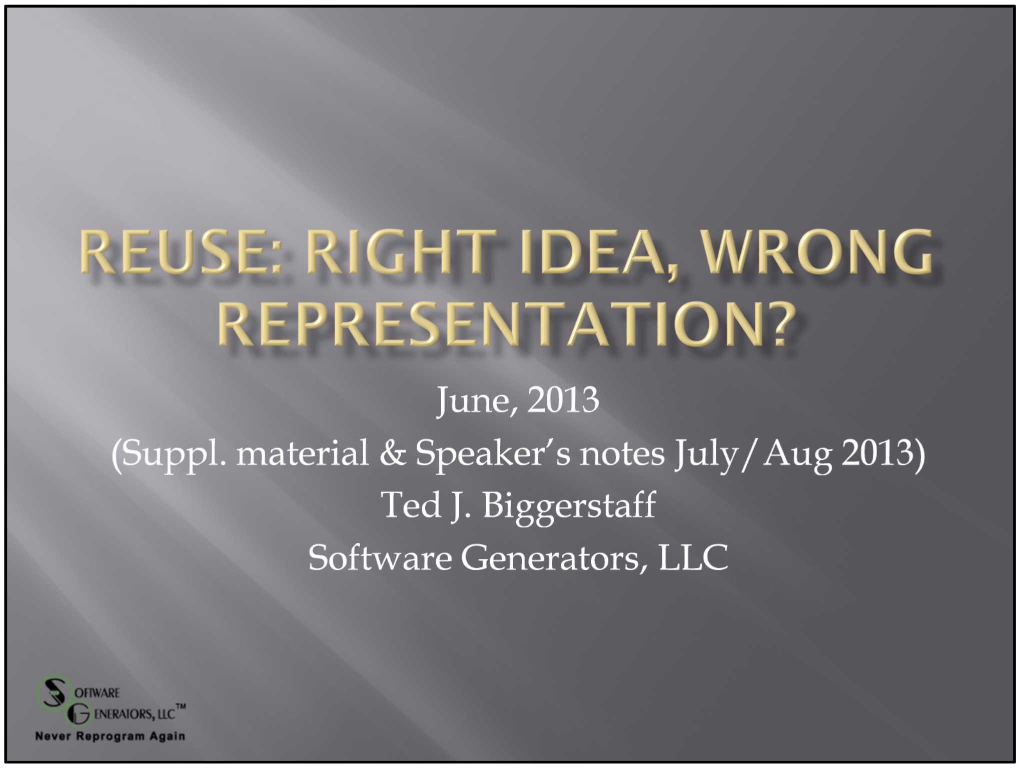


REUSE: RIGHT IDEA, WRONG REPRESENTATION?

June, 2013

(Suppl. material & Speaker's notes July/Aug 2013)

Ted J. Biggerstaff

Software Generators, LLC

This talk describes **DSLGen™** and its technology for translation of Domain Specific Languages (DSLs) into efficient code that exploits the high capability features (e.g., parallelism) of an arbitrary target execution machine. In principle, DSLGen™ can be applied to a computation in any problem domain and high capability code can be generated for any target machine. The examples focus on computations in the Digital Signal Processing (DSP) problem domain and illustrate generation of code for three different execution platforms (Von Neumann, multicore and vector).

The key idea underlying this technology is a **fundamental representational change** that completely differentiates it from previous work (i.e., from MDE, Refactoring, Aspect Oriented Programming, other synthesis work, etc.). That change rejects Programming Language (PL) Abstractions in the computational specification and even in the generator's early design process. In the initial specification of the computation and the initial representation of the generator's design of the implementation, there are no classes, methods, functions, scopes, or other abstractions that fall into the ***Programming Language Domain (PFD)***. Instead, the generator begins in the ***problem domain*** *by* using a ***fundamentally new abstraction*** -- an ***Associative Programming Constraint (APC)***. APCs factor out or isolate singular (possibly global) design features and thereby *partially* and *provisionally* specify each singular design feature in the eventual generated program. Only when a set of interrelated APCs are combined to form a Logical Architecture (LA) do they fully define how to map a domain specific expression of the computation into a generated program expressed in the PL domain (e.g., expressed in the C language). In the course of that overall generation process, the LA is evolved and reorganized by addition of so-called ***synthetic APCs*** that introduce elective design features (e.g., multicore parallelism and/or instruction level parallelism) into the final program. The mapping of the LA to final code additionally exploits ***design frameworks*** (a formalization of the "gang of four's" design patterns) to provide **global architectural frameworks** to house cloned and specialized elements of the computation specification (e.g., frameworks might include interrelated thread routine skeletons and global synchronization patterns) and **low level coding cliches** (e.g., portions of those frameworks might include thread cliches and individual synchronization steps)**.**

This talk provides a glimpse of what an APC is, how it is used, what a tools-eye view of the LA internal structures look like (i.e., what a domain engineer extending DSLGen™ might see) and what the code generated by DSLGen™ looks like.

Additionally, this fundamental representational change leads to a **fundamental operational change**. **Reprogramming is never required** to take advantage of new execution platform architectures and their specialized facilities. Only DSLGen™ needs a one time update with a new generator backend that tailors the generation process to the new execution platform.

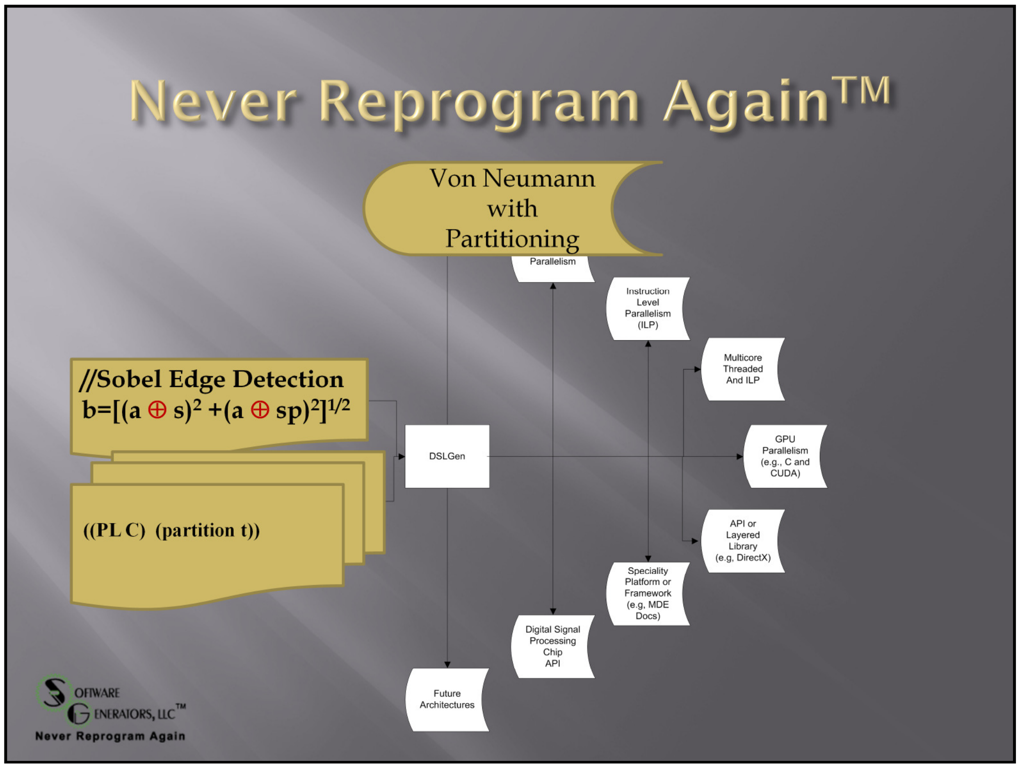
(Note: The kernel of this material is based on an invited paper (http://www.softwaregenerators.com/Papers/WorkshopPaper.pdf) and talk given at the DReMeR '13 workshop (http://www.nvc.cs.vt.edu/ICSRworkshop-DReMeR-13/technical-program.html) held in conjunction with 13th International Conference of Software Reuse (http://softeng.polito.it/ICSR13/index.html). Portions of this material were also used to give a Tools Demo at the main conference. Overall, this material provides a sneak peek at the forthcoming DSLGen™ generator system and its technology. This version of the material has been extended with additional slides, connecting links and speaker notes to allow a standalone understanding of the materials without the need to actually hear the live presentation.)

SOFTWARE GENERATORS, LLC™

Never Reprogram Again

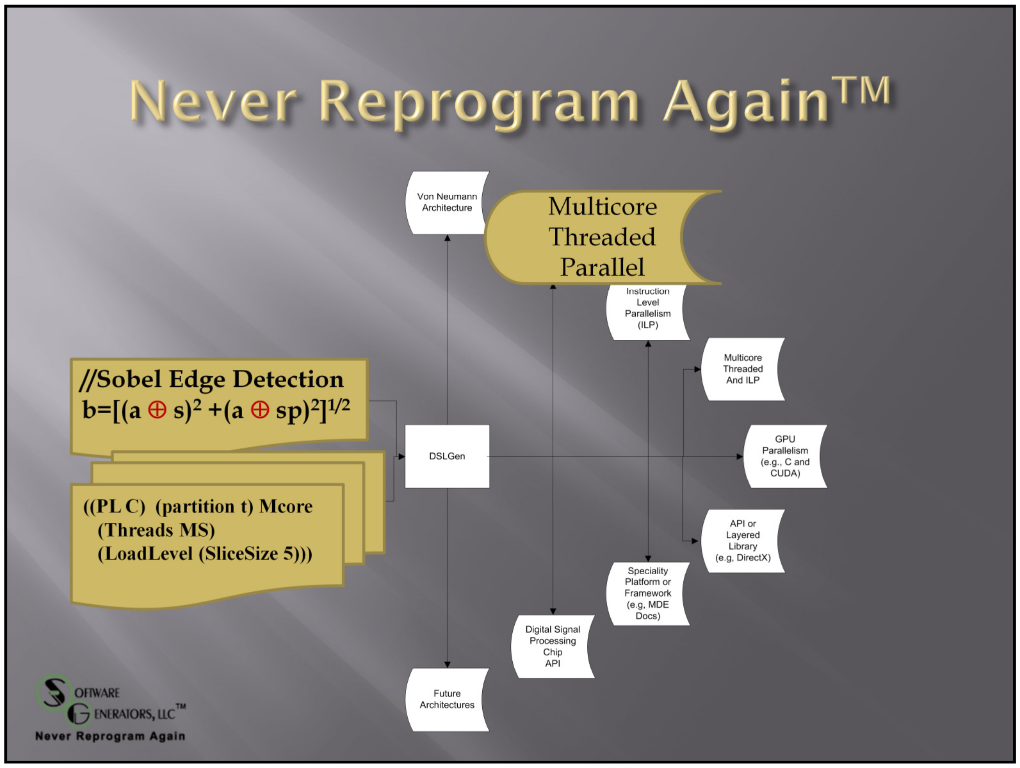
SOFTWARE GENERATORS, LLC™

Never Reprogram Again


Never Reprogram Again™
Von Neumann with Partitioning
Parallelism
Instruction Level Parallelism (ILP)
Multicore Threaded And ILP
//Sobel Edge Detection
$b=[(a \oplus s)^2 +(a \oplus sp)^2]^{1/2}$
DSLGen
GPU Parallelism (e.g., C and CUDA)
((PL C) (partition t))
API or Layered Library (e.g, DirectX)
Speciality Platform or Framework (e.g, MDE Docs)
Digital Signal Processing Chip API
Future Architectures
Software Generators, LLC™
Never Reprogram Again

The next few slides pictorially illustrate the broad goal of the DSLGen™ program generation system. It also illustrates several implementation architectures that can be handled by the current implementation of DSLGen™.

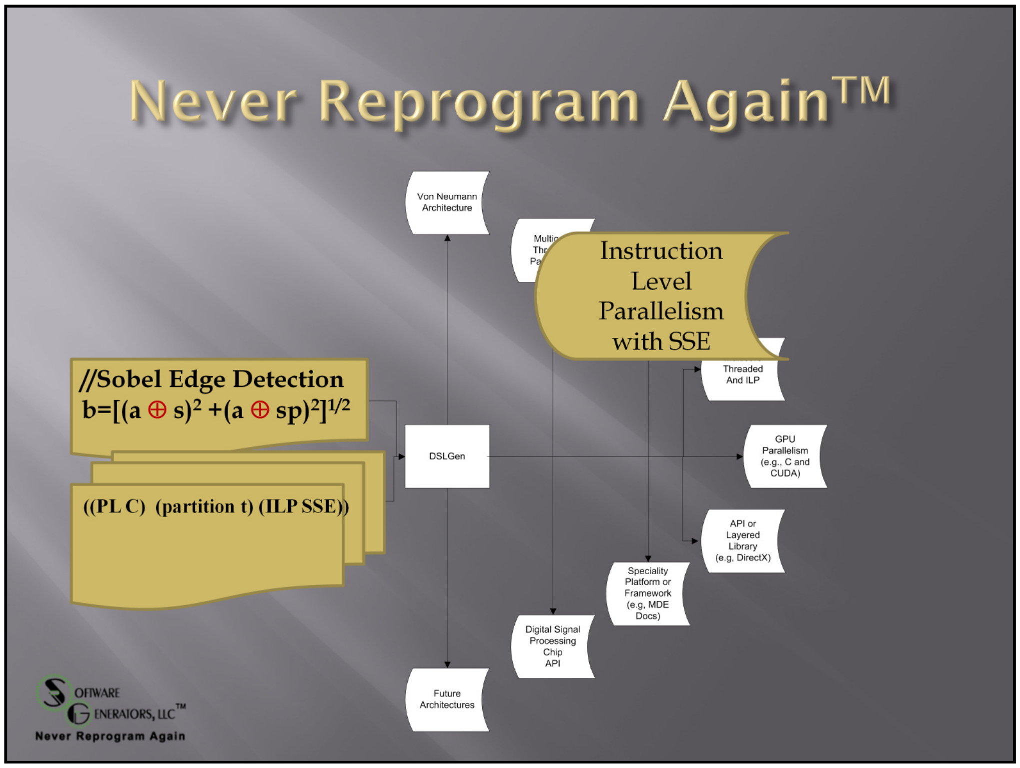
- Implementation neutral spec of computation ("Image Algebra", see Ritter et al) + separate spec of execution platform (both domain specific)
-Execution platform spec says emit C language and partitioned computation ("partitioning" to be defined later in this presentation)
- New execution platform architecture requires only new set of transformations, generator phases, and classes



Never Reprogram Again™

Von Neumann Architecture
Multicore Threaded Parallel
Instruction Level Parallelism (ILP)
Multicore Threaded And ILP
//Sobel Edge Detection
$b=[(a \oplus s)^2 +(a \oplus sp)^2]^{1/2}$
((PL C) (partition t) Mcore (Threads MS) (LoadLevel (SliceSize 5)))
DSLGen
GPU Parallelism (e.g., C and CUDA)
API or Layered Library (e.g, DirectX)
Speciality Platform or Framework (e.g, MDE Docs)
Digital Signal Processing Chip API
Future Architectures
Software Generators, LLC™
Never Reprogram Again

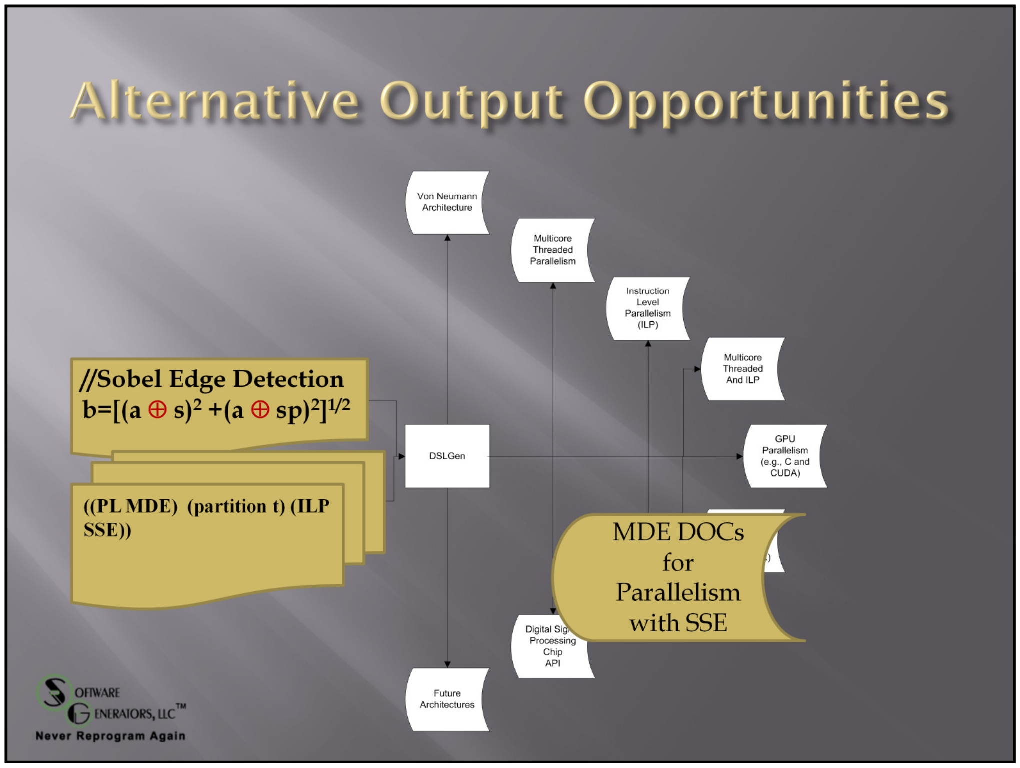
- Implementation neutral spec of computation UNCHANGED!!
- Spec for different execution platform (Optimize for multicore with MS threads and load leveling for heavyweight computations)



Never Reprogram Again™

Von Neumann Architecture
Multi Thr Pa
Instruction Level Parallelism with SSE
//Sobel Edge Detection
b=[(a ⊕ s)² +(a ⊕ sp)²]^1/2
((PL C) (partition t) (ILP SSE))
DSLGen
Threaded And ILP
GPU Parallelism (e.g., C and CUDA)
API or Layered Library (e.g, DirectX)
Speciality Platform or Framework (e.g, MDE Docs)
Digital Signal Processing Chip API
Future Architectures
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

- Implementation neutral spec of computation UNCHANGED!!
- Revised execution spec of execution platform (Optimize for Instruction level parallelism using Intel SSE instructions)



Alternative Output Opportunities

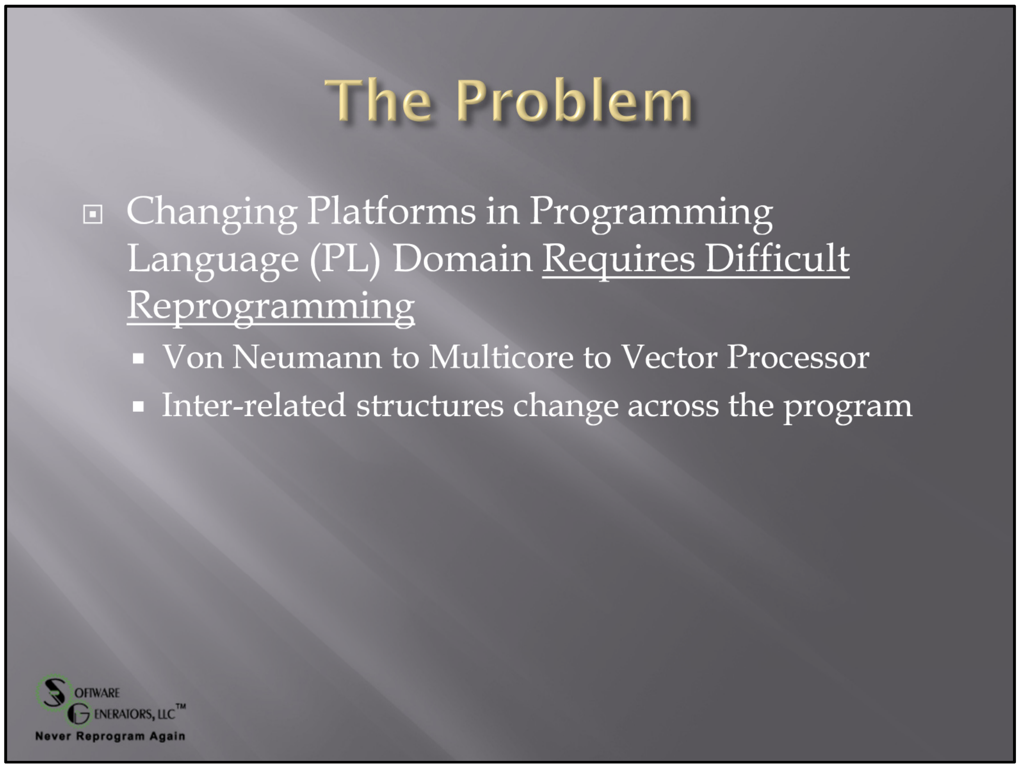
//Sobel Edge Detection
$b=[(a \oplus s)^2 + (a \oplus sp)^2]^{1/2}$

((PL MDE) (partition t) (ILP SSE))

MDE DOCs for Parallelism with SSE

Von Neumann Architecture
Multicore Threaded Parallelism
Instruction Level Parallelism (ILP)
Multicore Threaded And ILP
GPU Parallelism (e.g., C and CUDA)
DSLGen
Digital Sig Processing Chip API
Future Architectures
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

- Implementation neutral spec of computation UNCHANGED!!
- Revised execution spec of execution platform (Optimize for Instruction level parallelism using Intel SSE instructions and generate MDE artifacts)



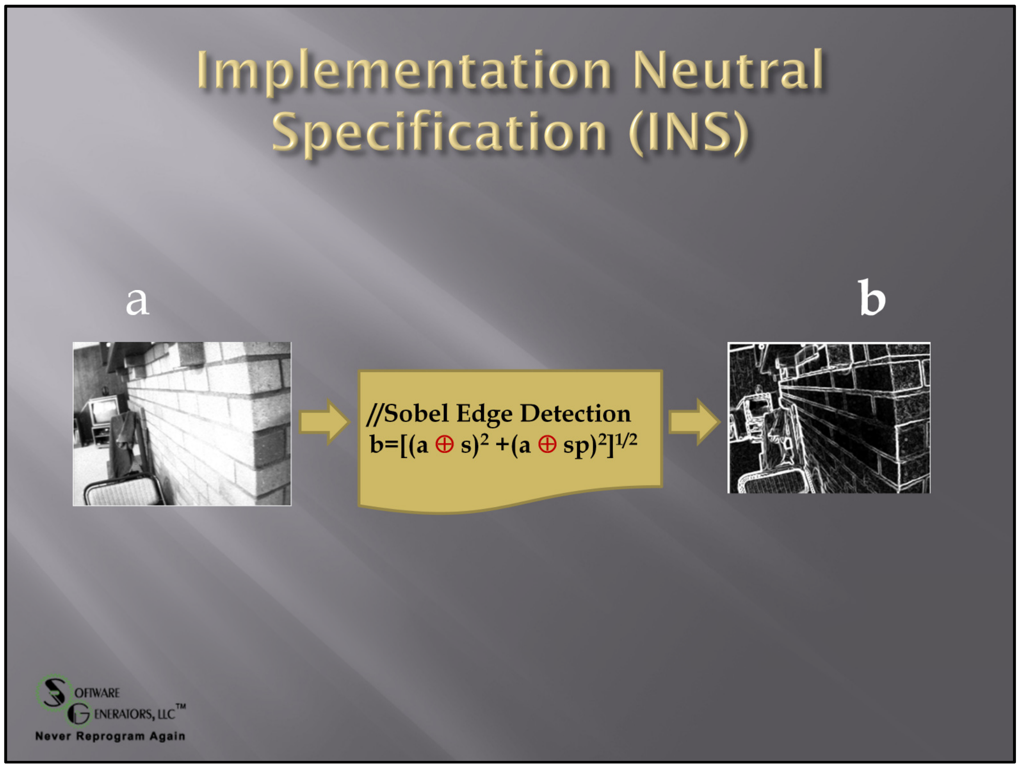
The Problem

- Changing Platforms in Programming Language (PL) Domain Requires Difficult Reprogramming
  - Von Neumann to Multicore to Vector Processor
  - Inter-related structures change across the program

OFTWARE
ENERATORS, LLC™
Never Reprogram Again

The problem with the goal of translating the domain specific (DS) specification of the computation into programming language (PL) implementations for a wide variety of differing execution architectures is that the PL implementations are widely varied in their forms and features, and there is no obvious way to create them from the DS specification. If we choose to express some progenitor form in a **PL-based representation** (even if the representation is abstracted as with MDE), then the transformation from that progenitor form to the varying implementation forms becomes a difficult and heretofore insolvable reprogramming problem.

Let us look at the variety of forms and features that must be created to see what we are up against.



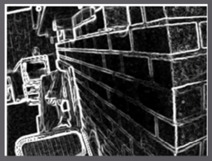
Implementation Neutral Specification (INS)

a

b

//Sobel Edge Detection

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

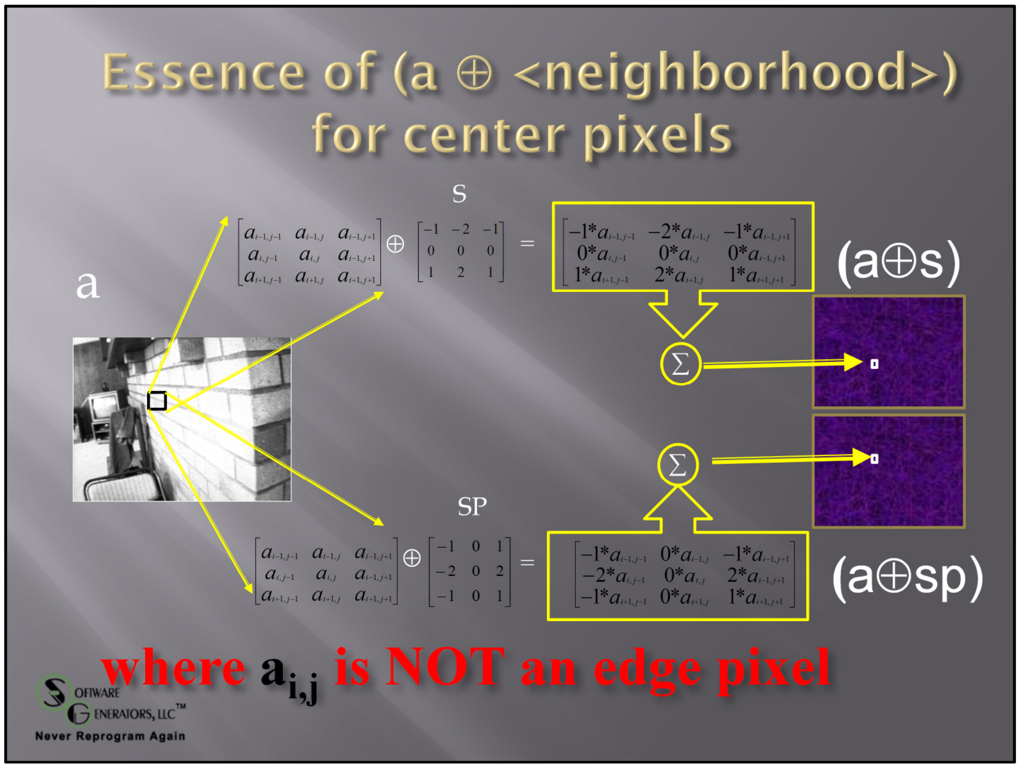


OFTWARE ENERATORS, LLC™ Never Reprogram Again

We will use a common example translation problem (Sobel based edge detection in images) that we will carry through the whole presentation. The images a and b illustrate a specific example of a Sobel edge detection computation. The specification of the Sobel edge detection computation is illustrated by the **Domain Specific (DS)** expression in the center, which is written in a DS language call the “Image Algebra” (IA) (See Ritter et al). The IA uses DS operators (e.g., ⊕, which is a “convolution”) operating on DS operands (e.g., images a and b and “neighborhoods” s and sp). We will define the convolution operation and a neighborhood in a minute. This expression is an **Implementation Neutral Specification (INS)** of the computation, which means that it is independent of the implementation architecture of the machine upon which it is to run on. That means that the INS never needs to be changed (i.e., to be reprogrammed) regardless of the existing or future implementation platform. It further means that the INS provides no information about implementation form that the computation will take when converted into code for some specific implementation architecture (e.g., multicore machine, vector machine or GPU based machine). The architecture of the implementation machine and the features of the machine that should be exploited will be separately specified via a set of high level DS descriptors (e.g., Multicore, ILP (Instruction Level Processing), SSE, etc.). Examples of these are illustrated in the first few slides.

So, what computation does this example INS expression specify? In summary, it says a grayscale (i.e., black and white) image a is “convolved” with a “neighborhood” s to produce a new image. The resultant image is then squared (i.e., each pixel value is squared). Then a second image is produced by convolving a with a different neighborhood sp and it is also squared. Those two images are added together (i.e., corresponding pixel values are added) producing a third image. Image b is produced by taking the square root of the third image.

Now, let’s look at the essence of this computation.



Essence of (a ⊕ <neighborhood>) for center pixels

$$\begin{bmatrix} a_{i-1,j-1} & a_{i-1,j} & a_{i-1,j+1} \\ a_{i,j-1} & a_{i,j} & a_{i-1,j+1} \\ a_{i+1,j-1} & a_{i+1,j} & a_{i+1,j+1} \end{bmatrix} \oplus \overset{s}{\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}} = \boxed{\begin{bmatrix} -1*a_{i-1,j-1} & -2*a_{i-1,j} & -1*a_{i-1,j+1} \\ 0*a_{i,j-1} & 0*a_{i,j} & 0*a_{i-1,j+1} \\ 1*a_{i+1,j-1} & 2*a_{i+1,j} & 1*a_{i+1,j+1} \end{bmatrix}}$$

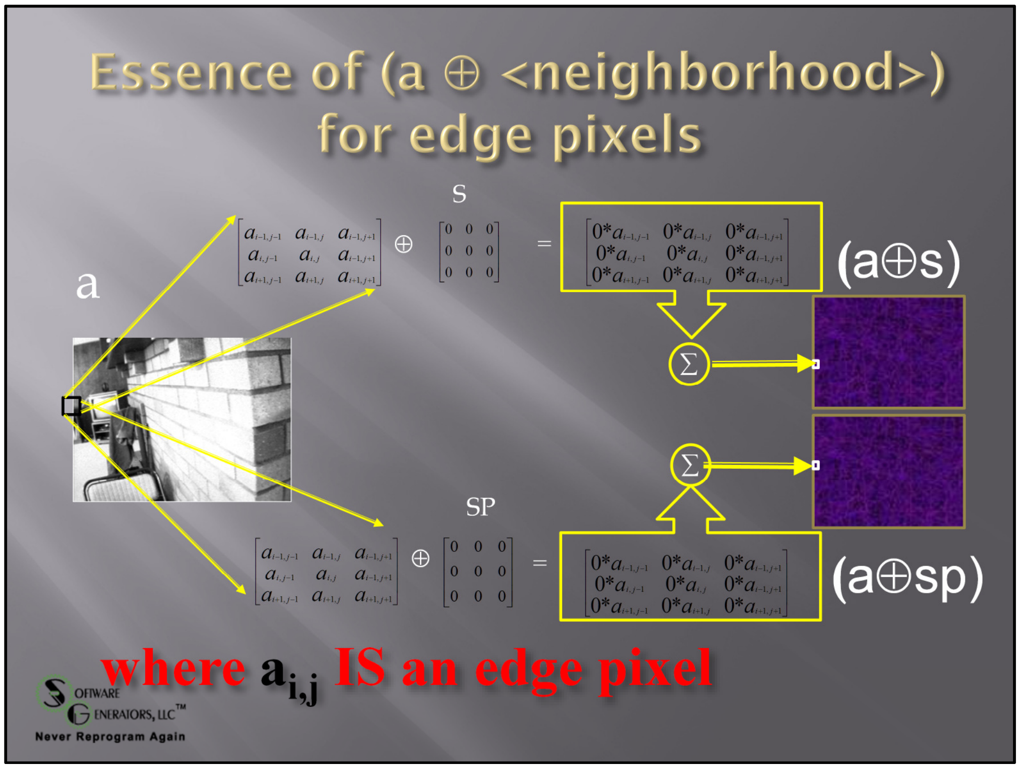
$$\begin{bmatrix} a_{i-1,j-1} & a_{i-1,j} & a_{i-1,j+1} \\ a_{i,j-1} & a_{i,j} & a_{i-1,j+1} \\ a_{i+1,j-1} & a_{i+1,j} & a_{i+1,j+1} \end{bmatrix} \oplus \overset{sp}{\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}} = \boxed{\begin{bmatrix} -1*a_{i-1,j-1} & 0*a_{i-1,j} & -1*a_{i-1,j+1} \\ -2*a_{i,j-1} & 0*a_{i,j} & 2*a_{i-1,j+1} \\ -1*a_{i+1,j-1} & 0*a_{i+1,j} & 1*a_{i+1,j+1} \end{bmatrix}}$$

where $\mathbf{a}_{i,j}$ is NOT an edge pixel

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

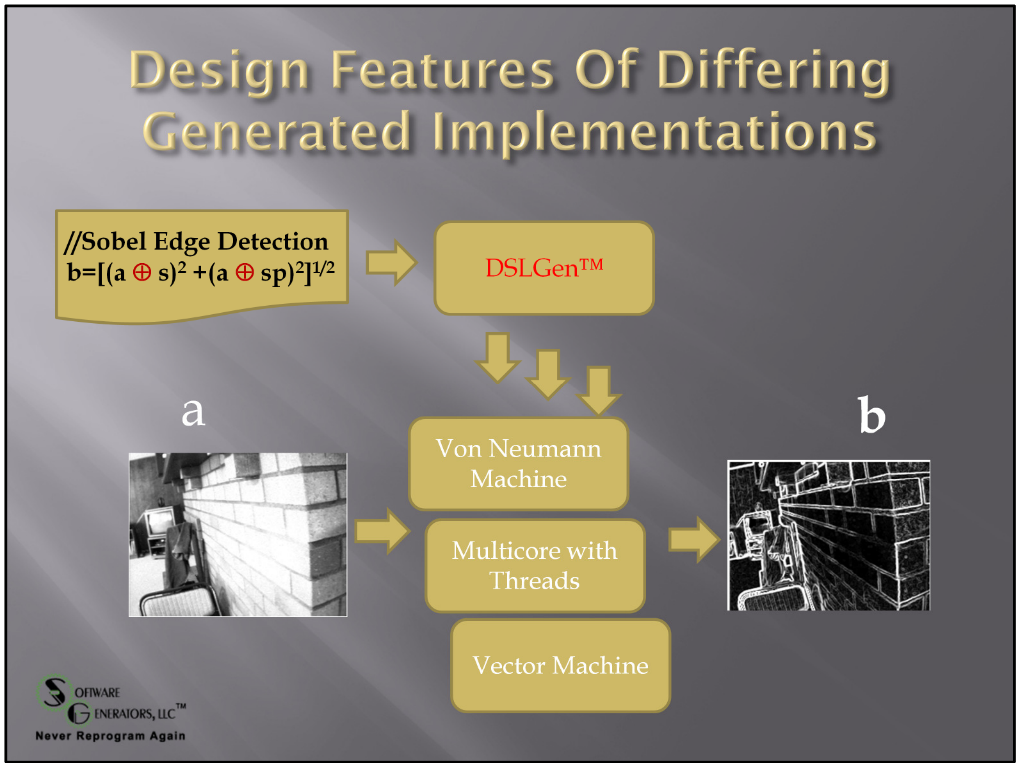
A convolution is computed for all non-edge (i.e., center) pixels a[i,j] by pair wise multiplying a[i,j] and its neighboring pixels with the corresponding weights of the neighborhood s and then summing all of those values, which produces a pixel at position [i,j] in the intermediate image (a convolve s). For the moment, we will informally represent the weights by some function w of s with "some args" to be defined later. The shorthand used for this will be "(w s …)". We will tighten up the definition of w later in this talk.

The neighborhood sp produces an analogous image. Notice that the weights of s are the weights of sp rotated 90 degrees clockwise, which is the hallmark of so-called Sobel edge detection. Remember these operations just produce the non-edge portions (i.e., center portions) of the first two intermediate images of the overall Sobel edge detection formula.


Essence of (a ⊕ <neighborhood>) for edge pixels
a
s
SP
(a⊕s)
(a⊕sp)
Σ
where $a_{i,j}$ IS an edge pixel
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

The case for <u>edge pixels</u> is analogous to the center pixel computation except that all weights of s and sp are 0 for edge pixels. Thus, the pixel value of (a convolve s) and (a convolve sp) will both be 0.

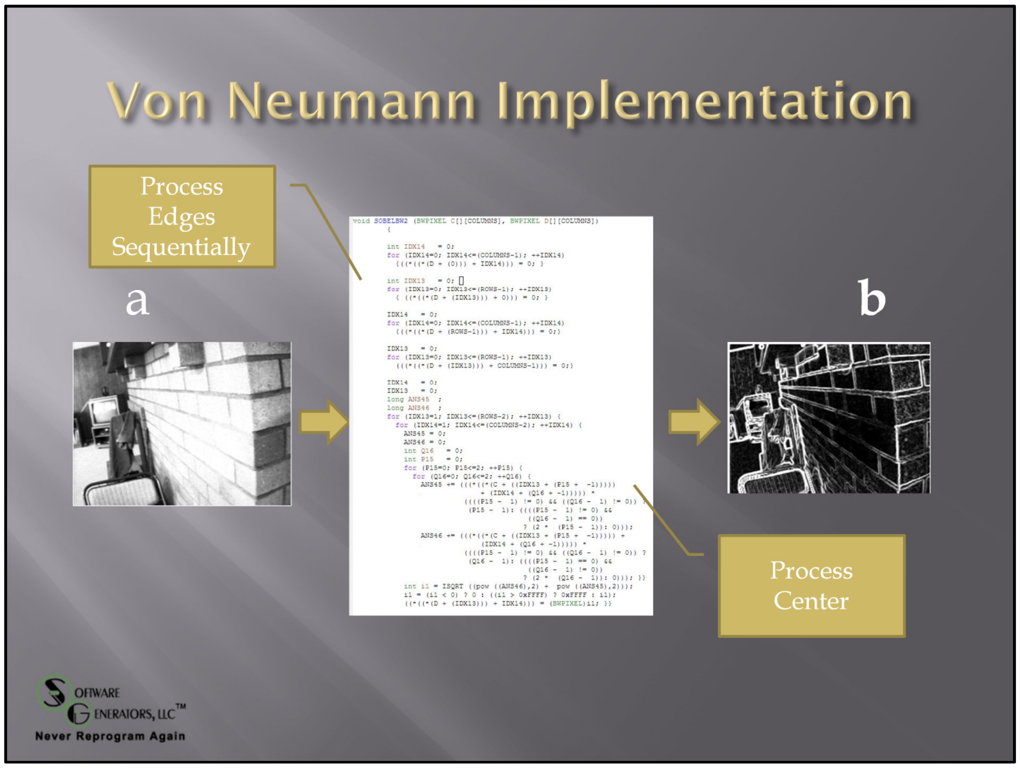
You may notice that we are being a bit sloppy with this chart illustrating the definition. How? Since a[i,j] is an edge pixel, at least one row or column of neighboring pixels will not exist (or a row <u>and</u> a column will be missing for corner pixels). The actual definitions DSLGen™ uses will account for this automatically but in the Sobel Edge detection definition (as opposed to some other computation, like image averaging), this is masked by the fact that (w s …) and (w sp …) are zero for edges. So, even if we did not properly account for this, partial evaluation would eliminate the error. However, it does matter in other computations, e.g., image averaging. In image averaging, the neighborhood loops have varying beginning and ending indexes that depend upon whether the pixel is a corner pixel (which will average four pixels), a non-corner edge pixel (which will average six pixels) or a center pixel (which will average nine pixels). These loop range variations will manifest themselves, in the implementation code generated, by a conditional expression for the indexes if the computation is <u>not partitioned</u> (i.e., the same code computes all cases) and by separated instances of neighborhood loops for the partitions (i.e., corners, non-corner-edges and centers) if the computation <u>is partitioned</u> (i.e., there are separate loops for each case). Bottom line: this presentation picture is not a fully precise expression of the underlying loop definitions but the generated code will be.



Design Features Of Differing Generated Implementations

//Sobel Edge Detection
$b=[(a \oplus s)^2 + (a \oplus sp)^2]^{1/2}$

Now, given those definitions let's see how these computations are expressed in implementations for different execution architectures. The important point is that we are now taking into consideration the structure of the execution architecture and exploiting what performance opportunities that it may provide. The next set of slides should provide insight into what the generator is up against in its goal of generating code customized to highly varied execution platforms. They also will provide concrete examples to motivate the generation steps that will be described as we illustrate how DSLGen™ operates.

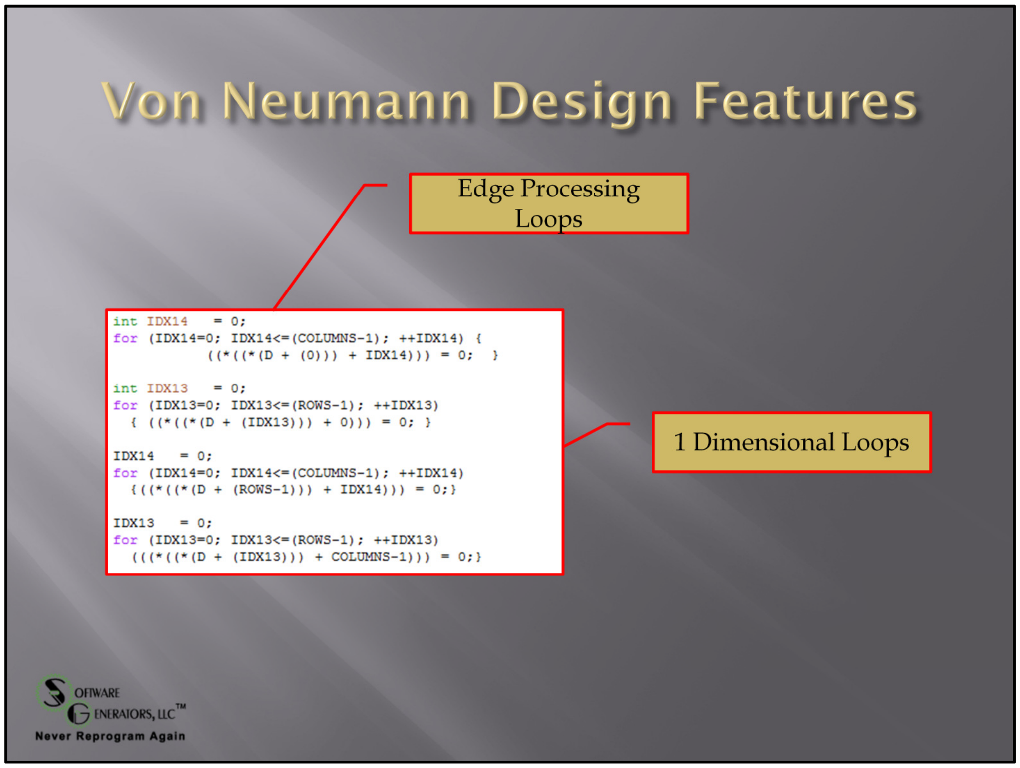


Von Neumann Implementation

Process Edges Sequentially
a
b
Process Center
OFTWARE ENERATORS, LLC™
Never Reprogram Again

```
void SOBELBW2 (BWPIXEL C[][COLUMNS], BWPIXEL D[][COLUMNS])
        {

        int IDX14   = 0;
        for (IDX14=0; IDX14<=(COLUMNS-1); ++IDX14)
          {((*((*(D + (0))) + IDX14))) = 0; }

        int IDX13   = 0; []
        for (IDX13=0; IDX13<=(ROWS-1); ++IDX13)
          { ((*((*(D + (IDX13))) + 0))) = 0; }

        IDX14   = 0;
        for (IDX14=0; IDX14<=(COLUMNS-1); ++IDX14)
          {((*((*(D + (ROWS-1))) + IDX14))) = 0;}

        IDX13   = 0;
        for (IDX13=0; IDX13<=(ROWS-1); ++IDX13)
          (((*((*(D + (IDX13))) + COLUMNS-1))) = 0;}

        IDX14   = 0;
        IDX13   = 0;
        long ANS45  ;
        long ANS46  ;
        for (IDX13=1; IDX13<=(ROWS-2); ++IDX13) {
          for (IDX14=1; IDX14<=(COLUMNS-2); ++IDX14) {
            ANS45 = 0;
            ANS46 = 0;
            int Q16   = 0;
            int P15   = 0;
            for (P15=0; P15<=2; ++P15) {
              for (Q16=0; Q16<=2; ++Q16) {
                ANS45 += (((*((*(C + ((IDX13 + (P15 +  -1)))))
                              + (IDX14 + (Q16 + -1))))) *
                          ((((P15 -  1) != 0) && ((Q16 -  1) != 0)) ?
                           (P15 -  1): ((((P15 -  1) != 0) &&
                                          ((Q16 -  1) == 0))
                                         ? (2 *  (P15 -  1)): 0)));
                ANS46 += (((*((*(C + ((IDX13 + (P15 +  -1))))) +
                               (IDX14 + (Q16 + -1))))) *
                          ((((P15 -  1) != 0) && ((Q16 -  1) != 0)) ?
                           (Q16 -  1): ((((P15 -  1) == 0) &&
                                          ((Q16 -  1) != 0))
                                         ? (2 *  (Q16 -  1)): 0))); }}
            int i1 = ISQRT ((pow ((ANS46),2) +  pow ((ANS45),2)));
            i1 = (i1 < 0) ? 0 : ((i1 > 0xFFFF) ? 0xFFFF : i1);
            ((*((*(D + (IDX13))) + IDX14))) = (BWPIXEL)i1; }}
```

For a Von Neumann machine, the generation is pretty much a matter in inlining domain operator and operand code definitions. Let’s look more closely at the design features of the simple Von Neumann implementation code…



Von Neumann Design Features

```
int IDX14   = 0;
for (IDX14=0; IDX14<=(COLUMNS-1); ++IDX14) {
          ((*((*(D + (0))) + IDX14))) = 0;  }

int IDX13   = 0;
for (IDX13=0; IDX13<=(ROWS-1); ++IDX13)
  { ((*((*(D + (IDX13))) + 0))) = 0; }

IDX14   = 0;
for (IDX14=0; IDX14<=(COLUMNS-1); ++IDX14)
  {((*((*(D + (ROWS-1))) + IDX14))) = 0;}

IDX13   = 0;
for (IDX13=0; IDX13<=(ROWS-1); ++IDX13)
  (((*((*(D + (IDX13))) + COLUMNS-1))) = 0;}
```

The only elective design feature introduced for this example is the restructuring of the computation such that it is "partitioned" into its natural case computational structure, i.e., the edge pixels are computed separately from the center pixels. Partitioning in this simple example is not of much performance value but it provides an opportunity to introduce the concept of partitioning to the audience within a simple context. In other examples, partitioning may be used in conjunction with other design features and patterns (e.g., thread based parallelism) to great performance advantage. In short, a partitioning separates a computation into separate pieces that are computed by different formulations of program code.

The "design feature" of note for edge pixel processing is that the generic 2 dimensional loops for processing a general pixel have degenerated to one dimensional loops and the pixel value computation (RHS of the assignment) has degenerated to a constant. As we will see in the tools demo (a later talk), this simplification will arise via an interaction between:

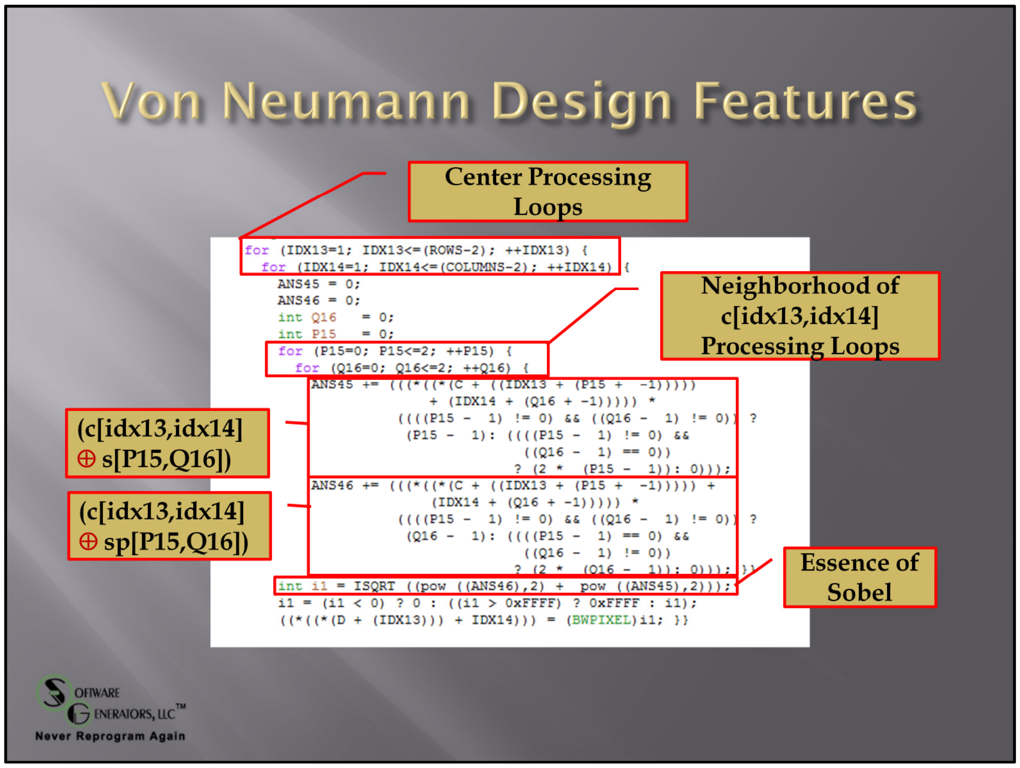
• partitioning conditions (e.g., (== Idx14 0) where Idx14 is one of the indexes),

• pixel computations specialized to a partitioning condition (e.g,. A neighborhood weight is 0 for an edge pixel),

• the inlining of domain specific component definitions, which is performed by the generator during a late phase, and

• the partial evaluation system, which will performs simplifications of the inlined expressions.

Now, let's look at the center pixel code.



Von Neumann Design Features

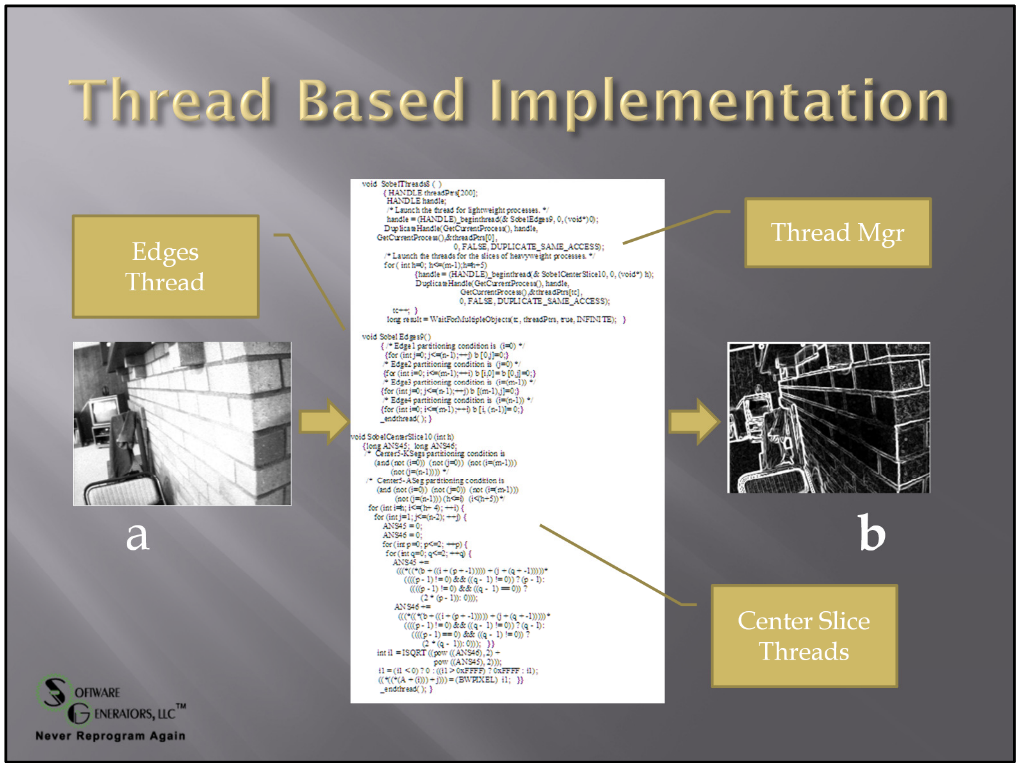
```
for (IDX13=1; IDX13<=(ROWS-2); ++IDX13) {
  for (IDX14=1; IDX14<=(COLUMNS-2); ++IDX14) {
    ANS45 = 0;
    ANS46 = 0;
    int Q16   = 0;
    int P15   = 0;
    for (P15=0; P15<=2; ++P15) {
      for (Q16=0; Q16<=2; ++Q16) {
        ANS45 += (((*((*(C + ((IDX13 + (P15 +  -1)))))
                       + (IDX14 + (Q16 + -1))))) *
                   ((((P15 -  1) != 0) && ((Q16 -  1) != 0)) ?
                    (P15 -  1): ((((P15 -  1) != 0) &&
                                  ((Q16 -  1) == 0))
                                 ? (2 *  (P15 -  1)): 0)));
        ANS46 += (((*((*(C + ((IDX13 + (P15 +  -1))))) +
                       (IDX14 + (Q16 + -1))))) *
                   ((((P15 -  1) != 0) && ((Q16 -  1) != 0)) ?
                    (Q16 -  1): ((((P15 -  1) == 0) &&
                                  ((Q16 -  1) != 0))
                                 ? (2 *  (Q16 -  1)): 0))); }}
    int i1 = ISQRT ((pow ((ANS46),2) +  pow ((ANS45),2)));
    i1 = (i1 < 0) ? 0 : ((i1 > 0xFFFF) ? 0xFFFF : i1);
    ((*((*(D + (IDX13))) + IDX14))) = (BWPIXEL)i1; }}
```

Center Processing Loops
Neighborhood of c[idx13,idx14] Processing Loops
(c[idx13,idx14] ⊕ s[P15,Q16])
(c[idx13,idx14] ⊕ sp[P15,Q16])
Essence of Sobel
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Here we see the generalized forms of the loops (i.e., two dimensions of processing) for the center pixels. But note that these loops clip off the edge pixels (as we would expect). That is, the main loops over the image have ranges of [1, (Rows -**2**)] and [1, (columns – **2**)] rather than [0, (Rows -**1**)] and [0, (columns – **1**)].

The loops processing the neighborhoods (P15 and Q16) are straightforward. Two answer variables have been introduced (ans45, ans46) and they correspond to the two convolution expressions in the problem domain specific specification of the Sobel computation. Later, we will see that the structure of the RHS of these computations is a pixel expression times a C language conditional expression that mimics the definition of the weight function (w) for either the s or sp neighborhood. Finally, the ISQRT expression mimics the form of the problem domain specific specification of the Sobel computation.

All in all, except for partitioning the edge pixel computations, the resultant code is not much more complex than code produced by simple inlining of definitions. However, that will change dramatically when we examine a thread based parallelism example next.

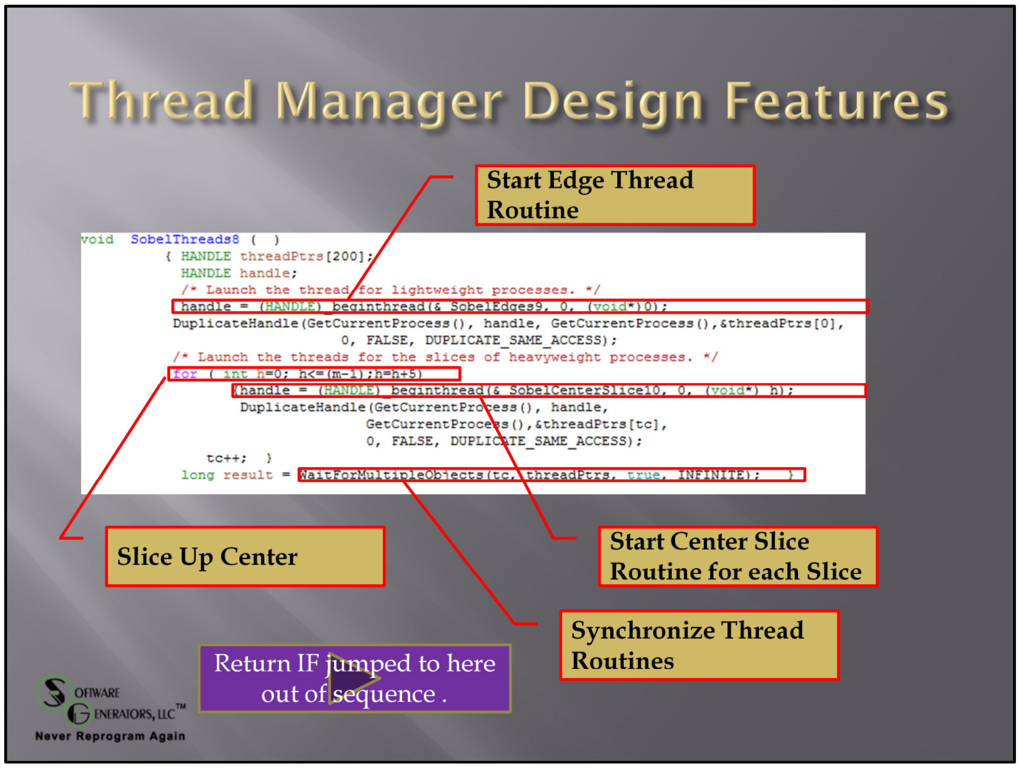


Thread Based Implementation

Edges Thread
Thread Mgr
Center Slice Threads
a
b
OFTWARE ENERATORS, LLC™
Never Reprogram Again

```
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);   }

void Sobel Edges9( )
    { /* 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]= b [0,j]=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  (i=(n-1)) */
    {for (int i=0; i<=(m-1);++i) b [i, (n-1)]= 0;}
    _endthread( ); }

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<(h+5)) */
   for (int i=h; i<=(h+ 4); ++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( ); }
```

For the case of thread based implementation on multicore machines, we see a **globally** different architecture in the generated code. We have three separate routines: a thread manager, a thread routine processing the edges and a thread routine for processing "slices" of the center partition. That is, the center partition has been further partitioned into slices that will be executed in parallel. So let's look at the thread manager's design features.



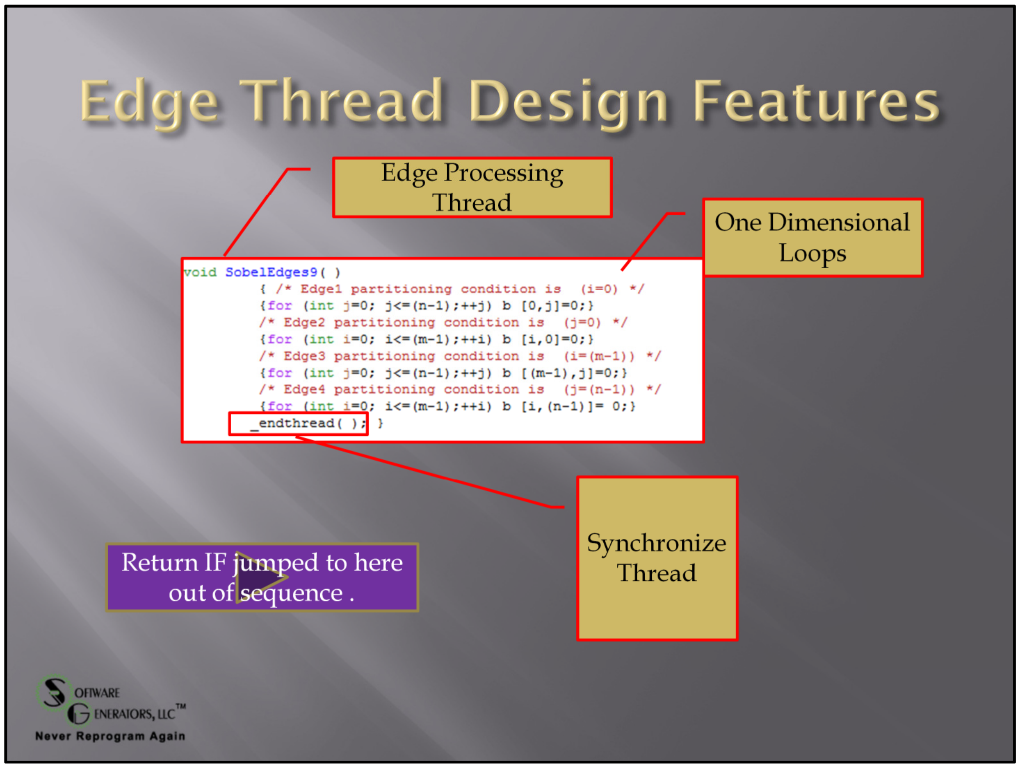
Thread Manager Design Features

```
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);   }
```

Start Edge Thread Routine
Slice Up Center
Start Center Slice Routine for each Slice
Synchronize Thread Routines
Return IF jumped to here out of sequence .
OFTWARE ENERATORS, LLC™
Never Reprogram Again

We see some rather low level detailed code, e.g., for starting up threads – one thread call for processing all of the edges, and one for starting each of the (many) center slice threads, which is inside a NEWLY minted loop that is calculating the starting index of the next center slice. The center slice is implied to be 5 rows of pixels. So, we infer that their will be floor ((m – 1)/5) slices and a partial slice if ((m – 1) mod 5) is not zero.

There is some other mysterious low level code, which we will ignore for this overview. In the generated code, the low level code detail arises from design frameworks. These will be introduced later in this presentation.

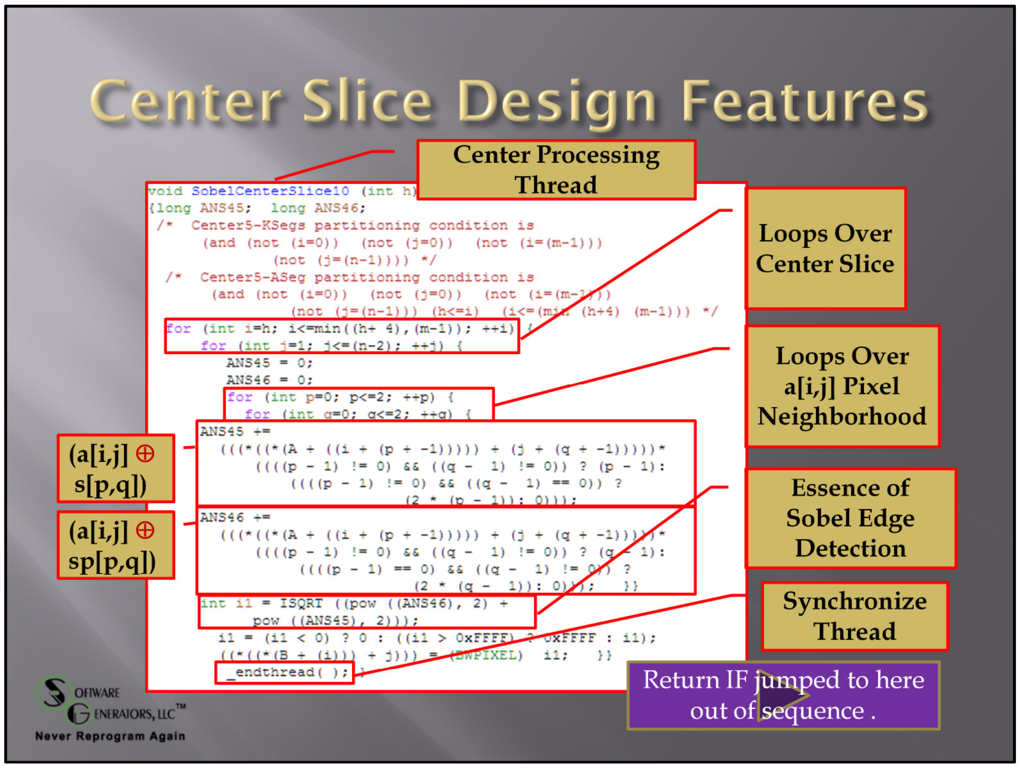


Edge Thread Design Features

```
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( ); }
```

Edge Processing Thread
One Dimensional Loops
Synchronize Thread
Return IF jumped to here out of sequence .
OFTWARE ENERATORS, LLC™
Never Reprogram Again

The edge pixel thread routine is pretty much like that of the Von Neumann example with the exception that there is some synchronization code added. It is important to note that the generator decided to batch up all of the edge pixel processing into a single thread routine. It uses domain specific knowledge about the typical computational heft of edge pixel computations to (heuristically) make this decision. A later example will illustrate this decision making.



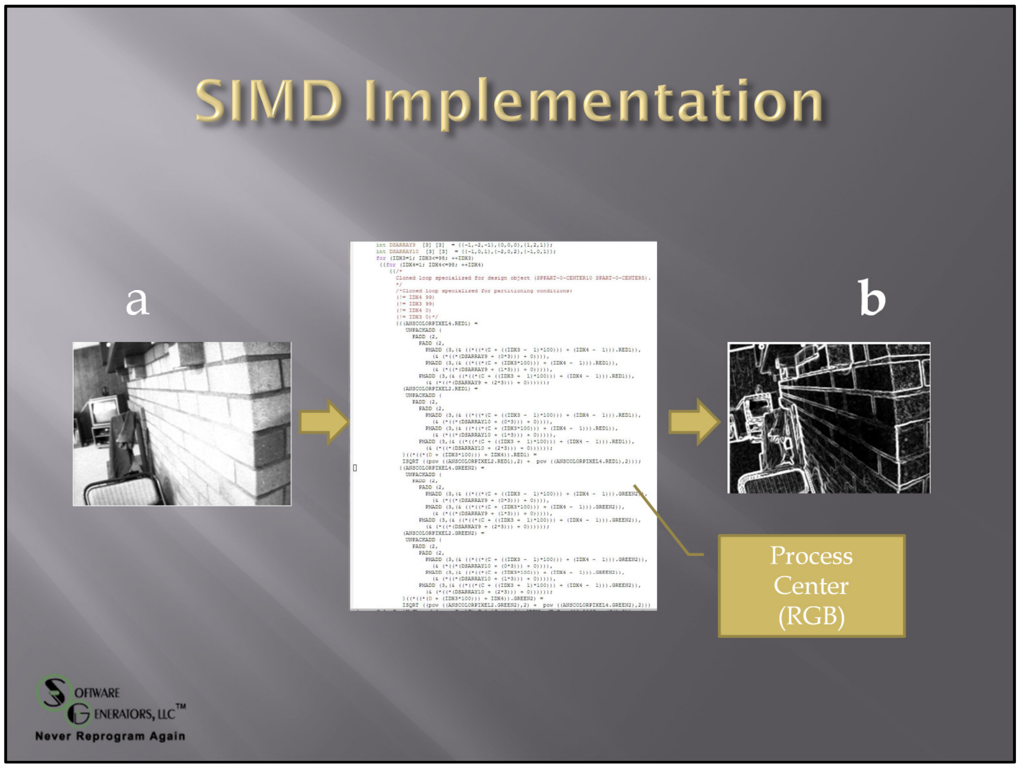
Center Slice Design Features

```
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 +=
  (((*((*(A + ((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 +=
  (((*((*(A + ((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);
  ((*((*(B + (i))) + j))) = (BWPIXEL)  i1;   }}
   _endthread( ); }
```

Center Processing Thread
Loops Over Center Slice
Loops Over a[i,j] Pixel Neighborhood
Essence of Sobel Edge Detection
Synchronize Thread
(a[i,j] ⊕ s[p,q])
(a[i,j] ⊕ sp[p,q])
Return IF jumped to here out of sequence .
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

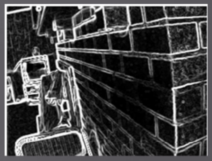
The thread routine that processes the center slice bears a striking although not exact resemblance to the Von Neumann center partition processing code. Notable differences are the thread synchronization code at the very end, the use of the h parameter as the starting row index for the slice and the min expression defining the loop completion. The min expression accounts for the (possibly) two different sizes for a center slice. That is, all but one slice are 5 rows of pixels and the last slice MAY be fewer than 5 rows. Or, it could also be 5 rows too.

On the other hand, if the user had specified the use of Instruction Level Processing (e.g., Intel's SSE instructions), there would be a much greater deviation in the form of the center processing code. So, let's take a look at code produced for ILP processing using SSE instructions.



SIMD Implementation

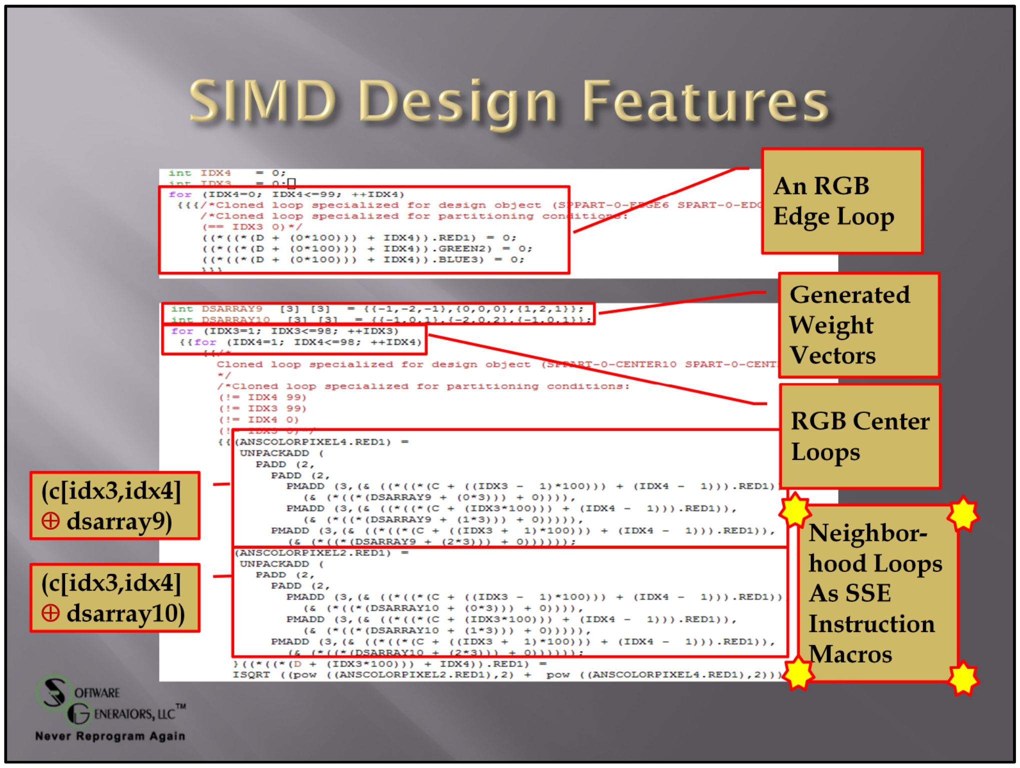
```
int DSARRAY9  [3] [3]  = {{-1,-2,-1},{0,0,0},{1,2,1}};
int DSARRAY10  [3] [3]  = {{-1,0,1},{-2,0,2},{-1,0,1}};
for (IDX3=1; IDX3<=98; ++IDX3)
 {{for (IDX4=1; IDX4<=98; ++IDX4)
    {{/*
      Cloned loop specialized for design object (SPPART-0-CENTER10 SPART-0-CENTER5).
      */
      /*Cloned loop 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)));
       ((ANSCOLORPIXEL4.GREEN2) =
         UNPACKADD (
           PADD (2,
             PADD (2,
               PMADD (3,(& ((*((*(C + ((IDX3 -  1)*100))) + (IDX4 -  1))).GREEN2)),
                 (& (*((*(DSARRAY9 + (0*3))) + 0)))),
               PMADD (3,(& ((*((*(C + (IDX3*100))) + (IDX4 -  1))).GREEN2)),
                 (& (*((*(DSARRAY9 + (1*3))) + 0))))),
             PMADD (3,(& ((*((*(C + ((IDX3 +  1)*100))) + (IDX4 -  1))).GREEN2)),
               (& (*((*(DSARRAY9 + (2*3))) + 0))))));
        (ANSCOLORPIXEL2.GREEN2) =
         UNPACKADD (
           PADD (2,
             PADD (2,
               PMADD (3,(& ((*((*(C + ((IDX3 -  1)*100))) + (IDX4 -  1))).GREEN2)),
                 (& (*((*(DSARRAY10 + (0*3))) + 0)))),
               PMADD (3,(& ((*((*(C + (IDX3*100))) + (IDX4 -  1))).GREEN2)),
                 (& (*((*(DSARRAY10 + (1*3))) + 0))))),
             PMADD (3,(& ((*((*(C + ((IDX3 +  1)*100))) + (IDX4 -  1))).GREEN2)),
               (& (*((*(DSARRAY10 + (2*3))) + 0))))));
        }((*((*(D + (IDX3*100))) + IDX4)).GREEN2) =
        ISQRT ((pow ((ANSCOLORPIXEL2.GREEN2),2) +  pow ((ANSCOLORPIXEL4.GREEN2),2)))
```



Process Center (RGB)

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

This is a slightly more complex example than the previous grayscale examples (i.e., it is for a color image with RGB color planes) but for the purposes of illustration, this example will illustrate the important design features nicely. The only major computational difference between this RGB example and a simple grayscale image example is that the Sobel processing code is duplicated for each color plane. However, the key implementation difference between this and previous examples is that the requirement to use SSE instructions induces significant changes to the design features of the code.



SIMD Design Features

```
int IDX4    = 0;
int IDX3    = 0;
for (IDX4=0; IDX4<=99; ++IDX4)
 {{{/*Cloned loop specialized for design object (SPPART-0-EDGE6 SPART-0-EDG
    /*Cloned loop specialized for partitioning conditions:
    (== IDX3 0)*/
    ((*(((*(D + (0*100))) + IDX4)).RED1) = 0;
    ((*(((*(D + (0*100))) + IDX4)).GREEN2) = 0;
    ((*(((*(D + (0*100))) + IDX4)).BLUE3) = 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-CENT
      */
      /*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)))
```

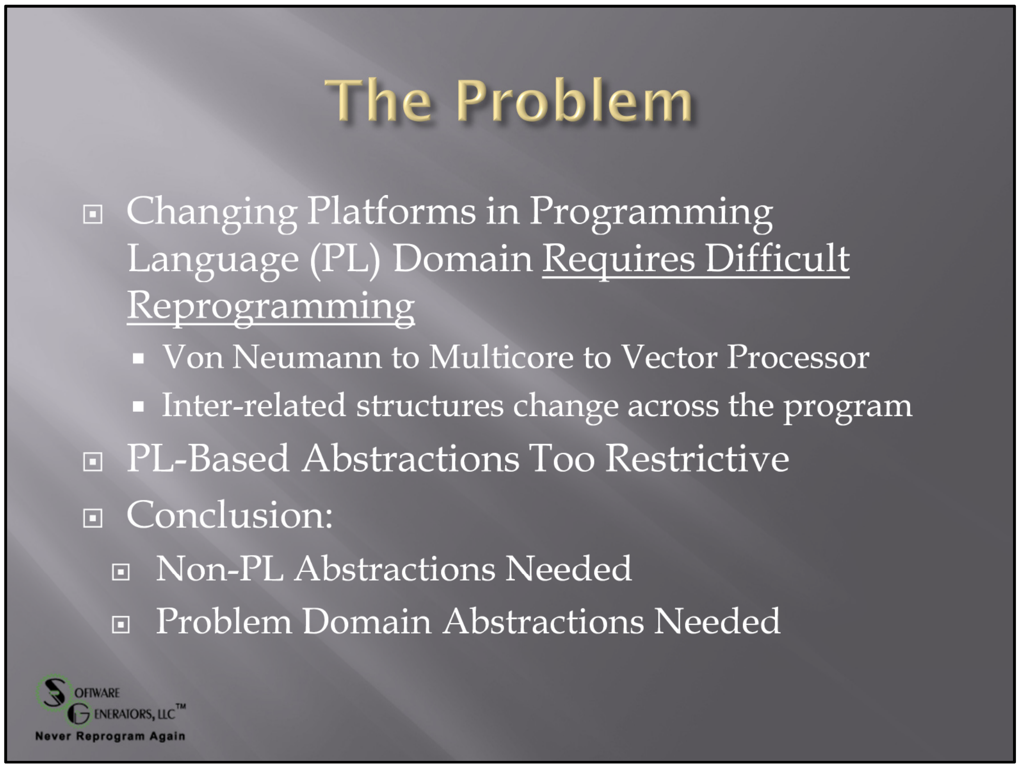
An RGB Edge Loop
Generated Weight Vectors
RGB Center Loops
(c[idx3,idx4] ⊕ dsarray9)
(c[idx3,idx4] ⊕ dsarray10)
Neighbor-hood Loops As SSE Instruction Macros
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

The edge pixel processing is duplicated for each color plane but other than that there is not any serious computational differences from the earlier examples. However, from the point of view of the implementation structure, there are significant global structural changes induced by the SSE design requirement in the center pixel processing code.

For the center pixel processing, only the red color plane processing is shown in this slide, but the green and blue processing is analogous. Notice the dsarray9 and dsarray10 are neighborhood weight arrays with the constant values that were shown in the earlier slide depicting the essence of the computation. The generator has constructed vectors of values because the SSE instruction that is most applicable to this computation (i.e., PMADD or product multiple and add) requires vectors as input. (BTW, the constant values are computed by the generator simulating a loop over the neighborhood positions and partially evaluating the definition of w for the specific neighborhood being processed.)

Importantly, notice that the neighborhood loops have completely disappeared and they have been replaced by expressions of calls to C macros (e.g., PMADD, PADD, and UNPACKADD) that will set up the data for and trigger the namesake SSE instructions. Each PMADD operation handles one row of neighborhood pixels and the corresponding neighborhood weight values from dsarray9 or dsarray10. The answer variables we saw earlier have become answer C structs (i.e., anscolorpixel2 and anscolorpixel4) that now have RGB fields.

For the record, if the user had specified both a thread based parallelism AND the use of SSE vector instructions, the code produced would be the expected integration of the two sets of design features.



The Problem

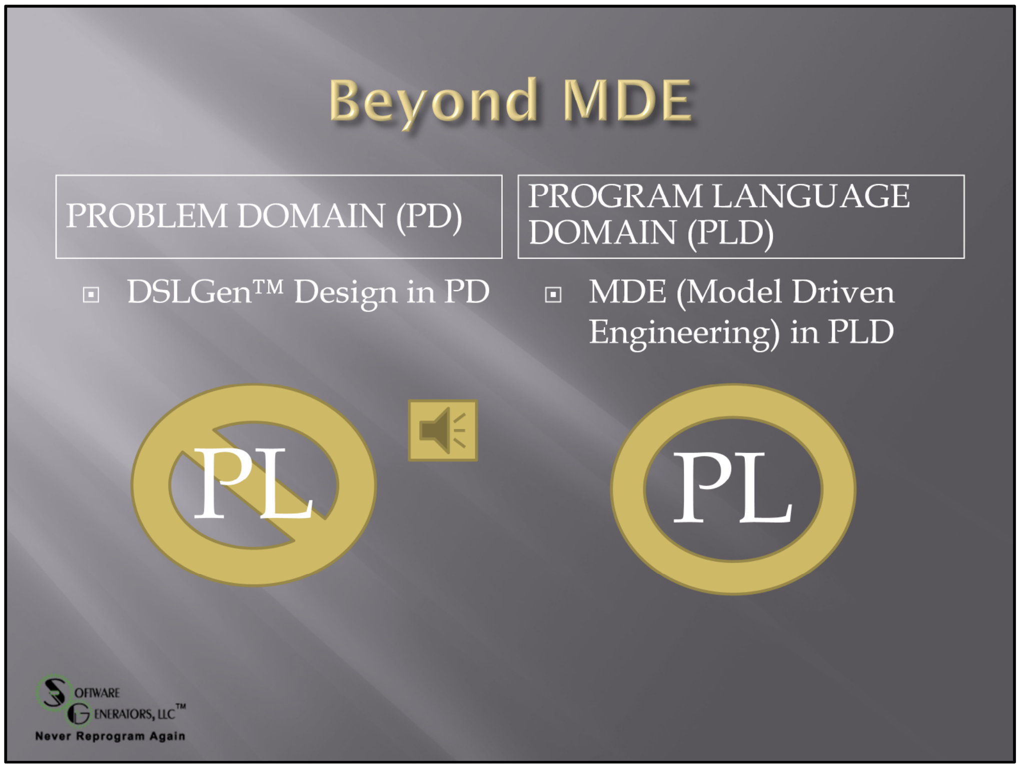
- Changing Platforms in Programming Language (PL) Domain Requires Difficult Reprogramming
  - Von Neumann to Multicore to Vector Processor
  - Inter-related structures change across the program
- PL-Based Abstractions Too Restrictive
- Conclusion:
  - Non-PL Abstractions Needed
  - Problem Domain Abstractions Needed

OFTWARE
ENERATORS, LLC™
Never Reprogram Again

So, the goal of this extended look at different implementation architectures for different execution platforms is 1) to illustrate the kind of code generated by DSLGen™ and 2) to disabuse the listener of the notion that it might be possible to automatically translate (i.e., reprogram) from one form to another. The programming language representation of these implementations makes automatic reprogramming an impossible problem (in my opinion). Even if one attempts to abstract the PL forms as with Model Driven Engineering, the problem is still too difficult to accomplish for all but the most trivial problems. The lack of applications for purchase that will perform such reprogramming jobs is prima facie evidence for this conclusion. Most reprogramming tools that one can purchase have a very narrow focus and have **a smart human programmer** firmly in the processing loop.

This conclusion has driven the DSLGen™ work away from PL based representations and toward non-PL, problem domain oriented representations. And this is key to DSLGen™ 's ability to generate optimized code for such a diverse and opened ended set of execution architectures.

So, let's examine this idea of problem domain abstractions and see how they work.

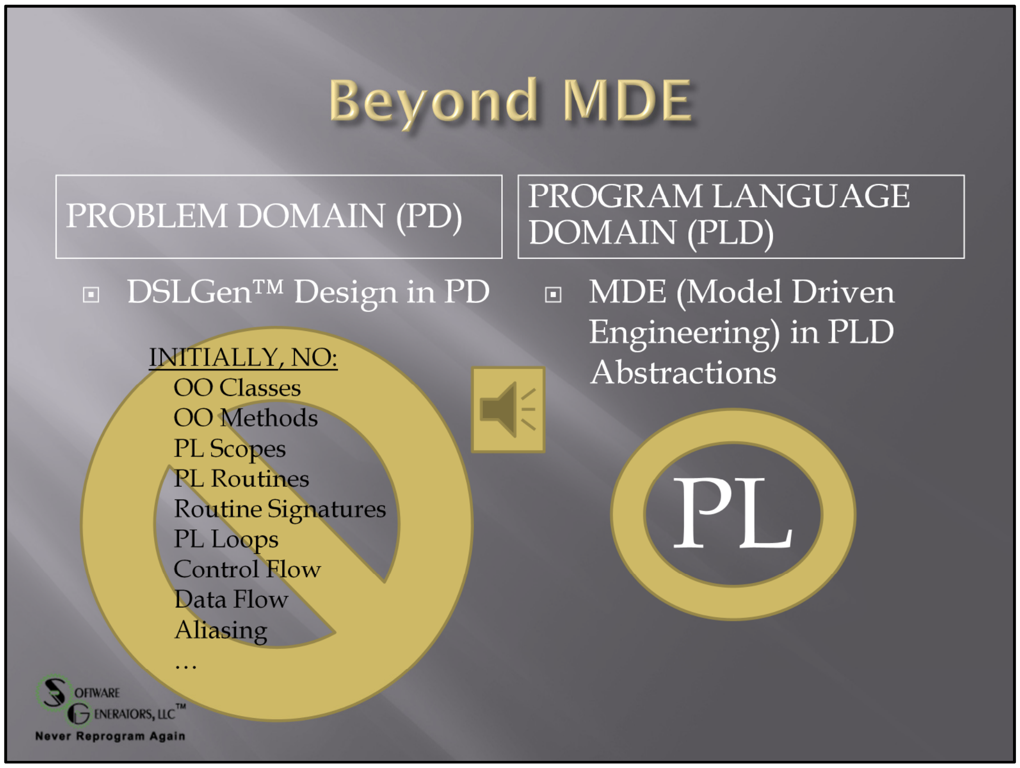


Beyond MDE

PROBLEM DOMAIN (PD)	PROGRAM LANGUAGE DOMAIN (PLD)
DSLGen™ Design in PD	MDE (Model Driven Engineering) in PLD

PL
PL
OFTWARE
ENERATORS, LLC™
Never Reprogram Again

The DSLGen™ model rejects programming language abstractions (PL) as being too restrictive for the early design phases and invents a fundamentally new abstraction that is specific to the problem domain. DSLGen™ even eschews MDE abstractions and models (even though MDE's objective is to make the program specification more generic and therefore, more easily changeable) because those abstractions and models are still are based on abstractions that are PL oriented (e.g., OO classes, procedures, parametric communication, etc.).   Even though MDE labors to abstract the PL structures, it still allows, and in a sense, forces the engineer to commit to models that are biased toward one or another particular implementation architecture (e.g., threaded parallelism or vector parallelism). This bias may make choosing one implementation form easy but a different one difficult. So, even the MDE level of abstraction is problematic for generation for a wide range of different implementation architectures.



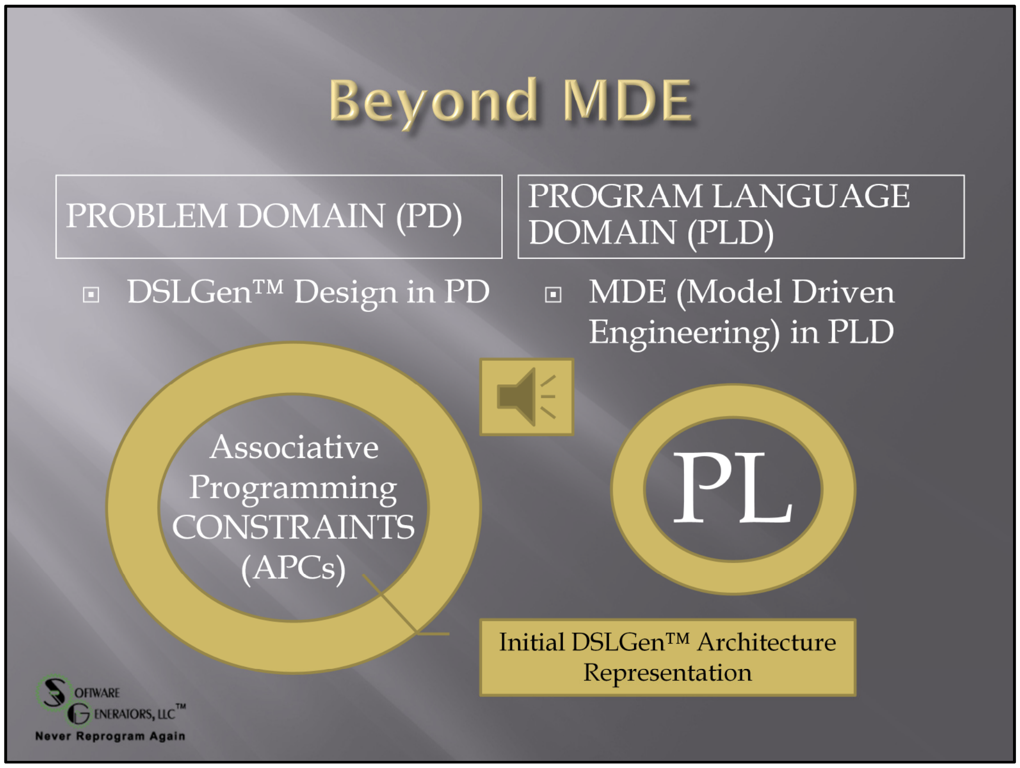
Beyond MDE

PROBLEM DOMAIN (PD)	PROGRAM LANGUAGE DOMAIN (PLD)
DSLGen™ Design in PD	MDE (Model Driven Engineering) in PLD Abstractions

INITIALLY, NO:
- OO Classes
- OO Methods
- PL Scopes
- PL Routines
- Routine Signatures
- PL Loops
- Control Flow
- Data Flow
- Aliasing
- …

PL
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

This slide just makes the point a bit more concretely. Abstractions that define "how" the computation proceeds or "how" it is architected are too strongly committal to one particular implementation architecture or another. The elements that DSLGen™ avoids are those that define the "how" of structures whose concrete forms will eventually appear in the final code – things such as OO classes and methods, routines, loops, control flow, etc. We want to leave the DERIVATION of those elements in the generated code to the generator because that allows greater possible variation in the final product. And as important, or maybe even more important, it eliminates the need to reprogram the target program when it needs to be rehosted on an implementation platform with a significantly different architecture, one that requires significantly different design structures to exploit that architecture.



Beyond MDE

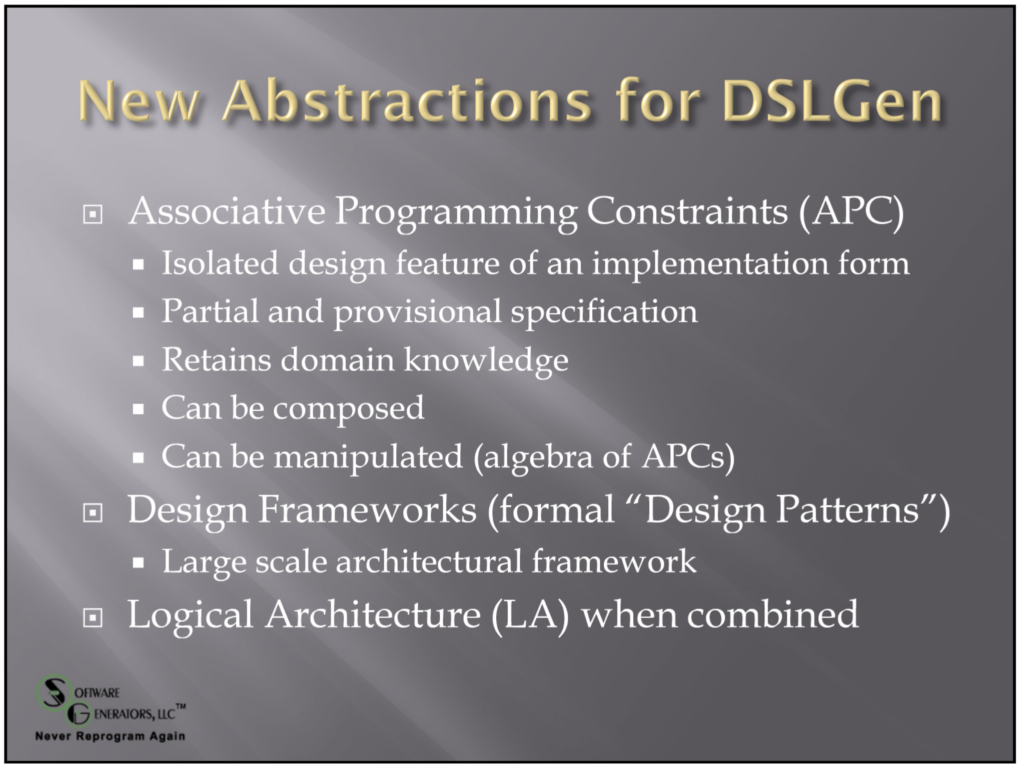
PROBLEM DOMAIN (PD)	PROGRAM LANGUAGE DOMAIN (PLD)
▣ DSLGen™ Design in PD	▣ MDE (Model Driven Engineering) in PLD

Associative Programming CONSTRAINTS (APCs)
PL
Initial DSLGen™ Architecture Representation
OFTWARE ENERATORS, LLC™
Never Reprogram Again

So, rather than build explicit structures that are abstract forms of PL structures, DSLGen™ builds structures of constraints that "imply" design features of the eventual PL structures that will be derived. These constraints are called <u>Associative Programming Constraints</u> (or <u>APCs</u>) because they are associated with the domain specific computation elements much like natural language modifiers are associated with the grammatical elements that they modify. APCs are <u>problem domain objects</u> (in the OO sense) that modify domain specific expressions of a computation. Each APC identifies one singular (possibly global) design feature that will be in the eventual implementation code of that computation. Can you write code from an APC? <u>NO</u>! You need a whole set of APCs that provide all of the design features that will be in the final implementation in order to generate implementation code. This is because there is <u>rarely a one to one mapping between APCs and PL-based abstractions of the implementation code</u>. In general, the mapping is a many to many mapping.

If you take one idea away from this talk, <u>it should be that APCs are not Programming Language based implementation structures</u>. <u>They are not executable forms</u>.

APCs are a <u>fundamentally new abstraction</u> that makes generating a highly varying range of implementation forms a solvable problem. Let's look a little more closely at APCs.



New Abstractions for DSLGen

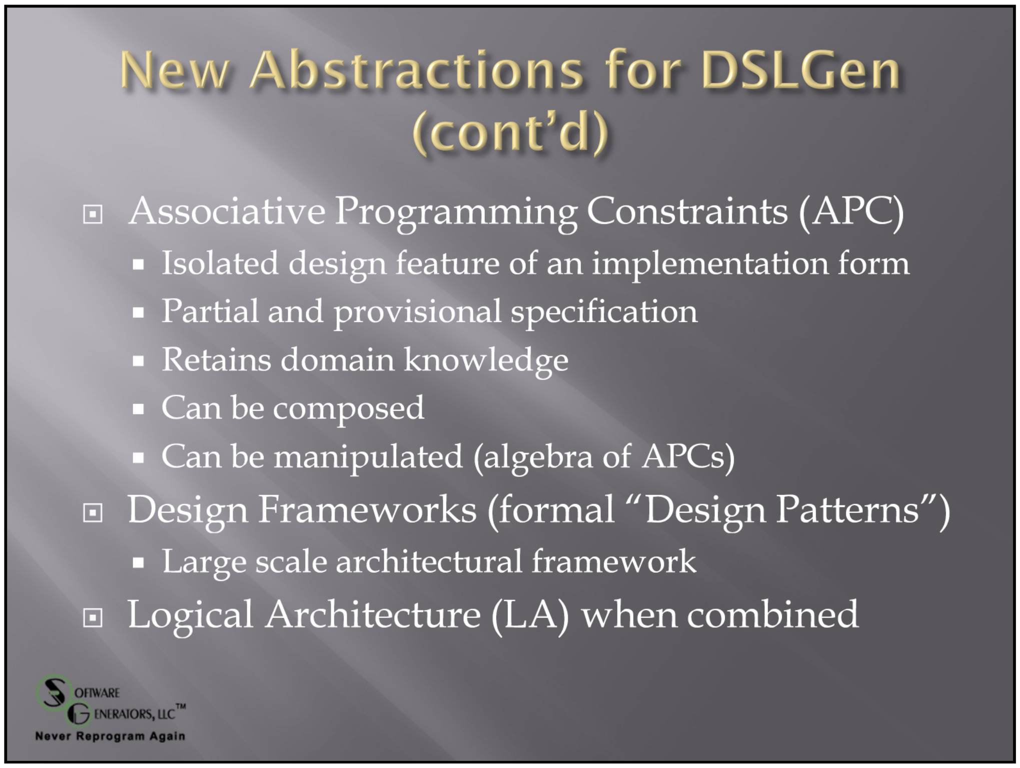
- Associative Programming Constraints (APC)
  - Isolated design feature of an implementation form
  - Partial and provisional specification
  - Retains domain knowledge
  - Can be composed
  - Can be manipulated (algebra of APCs)
- Design Frameworks (formal "Design Patterns")
  - Large scale architectural framework
- Logical Architecture (LA) when combined

Software Generators, LLC™
Never Reprogram Again

The motivation for APC's is analogous to the motivation for modifiers in natural language. That is, an APC is a modifier of a domain specific expression (e.g., a convolution expression) that specifies some distinct design feature in the eventual programming language (PL) implementation form of that domain specific expression. For example, the APC might provide the "nominal" form of the loop or loops required to perform the computation. However, this is only a partial specification of the PL implementation form because it does not determine the context or even the concrete implementation form of the loops. There are many open questions unanswered by a singular APC. For example: Are the implementation loop or loops partitioned into pieces? And are those loop pieces organized into thread routines or re-expressed as SSE instructions (e.g., PMADD instructions)? If SSE instructions, what triggers the reorganization necessary to reform the weight values into vectors (e.g., DSArray9 and DSArray10 of pervious example)? And so forth. Thus a singular APC is unlikely to be sufficient to generate the implementation for the target routine. It is unlikely that code written directly from a singular APC will be the same as the eventual code of the generated target routine. Too many other design features (represented by other APCs) will be needed to fulfill the requirements imposed by the user's description of the execution platform features to be exploited in the target routine. It is much more likely that a number of APCs will be required to fully specify the PL implementation. Moreover, like modification structures in natural language, these APCs will need to be formulated into a structure (i.e., a logical architecture or LA) that captures the interrelationships among them. For example, a partitioning APC may modify a loop APC and therefore imply addition features of the PL loop implementation.

Furthermore, the APCs are provisional in that the LA is likely to change as the generator introduces new synthetic APCs that introduce new elective design features (e.g., repartitionings of loops) and as it revises the structure of the LA. For example, component definitions may be specialized to and therefore organized under the new partitions. That is, the structural relationships of the LA evolve step by step toward the structural relationships that will exist in the final PL implementation form.

Conventionally, one tends to think of "constraints" as being represented by some kind of formulaic expression (e.g., a predicate calculus expression). However, while formulaic expressions do play a role in some APCs (e.g., partition APCs will have a so-called "partitioning condition" expression), APCs also have several additional representational facets and features. For the most part, they are CLOS objects that imply something about the eventual PL implementation by their existence and interrelationships. But beyond that, because they are problem domain entities, they also may have domain knowledge features or properties. For example, image "edges" have the domain property of being "lightweight" computations and that property may be employed by the generator to decide upon thread designs. Recall that the edge loops in the earlier thread design were batched into a single thread rather than each having their own thread. The "lightweight" domain property was used heuristically by the generator to make that design decision. Similarly, image center partitions are known to be "heavyweight" computations because there are often many individual computations to be done and because the individual computations are often fairly complex. That domain property was used by the generator to decide to slice the center partition into computational slices and to assign each to its own parallel thread.


New Abstractions for DSLGen (cont'd)

- Associative Programming Constraints (APC)
  - Isolated design feature of an implementation form
  - Partial and provisional specification
  - Retains domain knowledge
  - Can be composed
  - Can be manipulated (algebra of APCs)
- Design Frameworks (formal "Design Patterns")
  - Large scale architectural framework
- Logical Architecture (LA) when combined

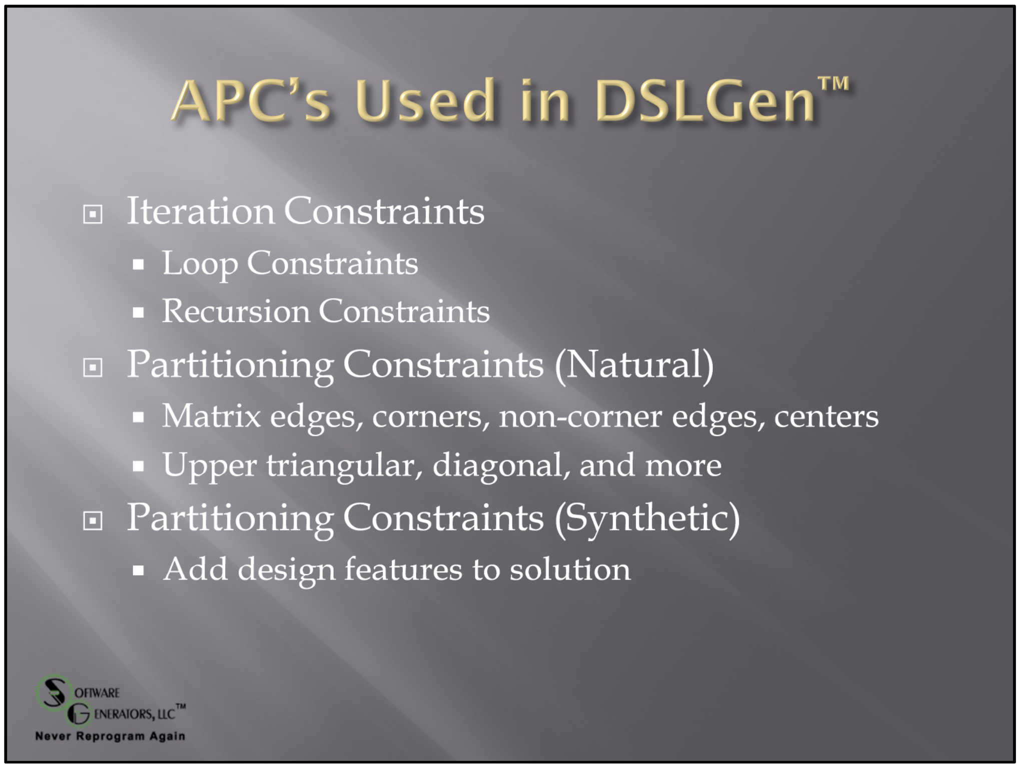
(This hidden slide allows continuation of speaker's notes for this slide.)
APCs are composed and manipulated to represent the evolving design of the final target program. The "algebra" of these compositions and manipulations for a particular problem domain is expressed by the transformations that define the processing of DSLGen™ for domain specific expressions in that domain.

While partitioned computations (defined via partitioning APCS) represent the essential pieces of the overall computation, they provide little or no information about what kind of a framework will be required by the PL context to allow these essential pieces to do their work and then integrate their individual results to realize a correct and coherent computation. This information will be provided by a separate abstraction, a design framework, which will be uniquely determined by the data and form of the LA in conjunction with the user's description of the target implementation platform. The design framework will provide both

1. **A Global architectural framework** to house the cloned and specialized elements of the computation specification (e.g., frameworks might include interrelated thread routine skeletons and global synchronization patterns) and
2. **Low level coding clichés** specific to the design framework (e.g., portions of those frameworks might include thread clichés and individual synchronization steps)**.**

One can think of a design framework as a formalization of the "Gang of four's" notion of a Design Pattern.

Design frameworks complete the implied design of the target computations. Later in this presentation, we will examine design frameworks more closely in the course of walking through the overall generation process for a thread based implementation for a multicore machine. But for the moment, let's look a bit more closely at the kinds and semantics of the APCs the generator is dealing with. **..next slide..**



APC's Used in DSLGen™

- Iteration Constraints
  - Loop Constraints
  - Recursion Constraints
- Partitioning Constraints (Natural)
  - Matrix edges, corners, non-corner edges, centers
  - Upper triangular, diagonal, and more
- Partitioning Constraints (Synthetic)
  - Add design features to solution

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

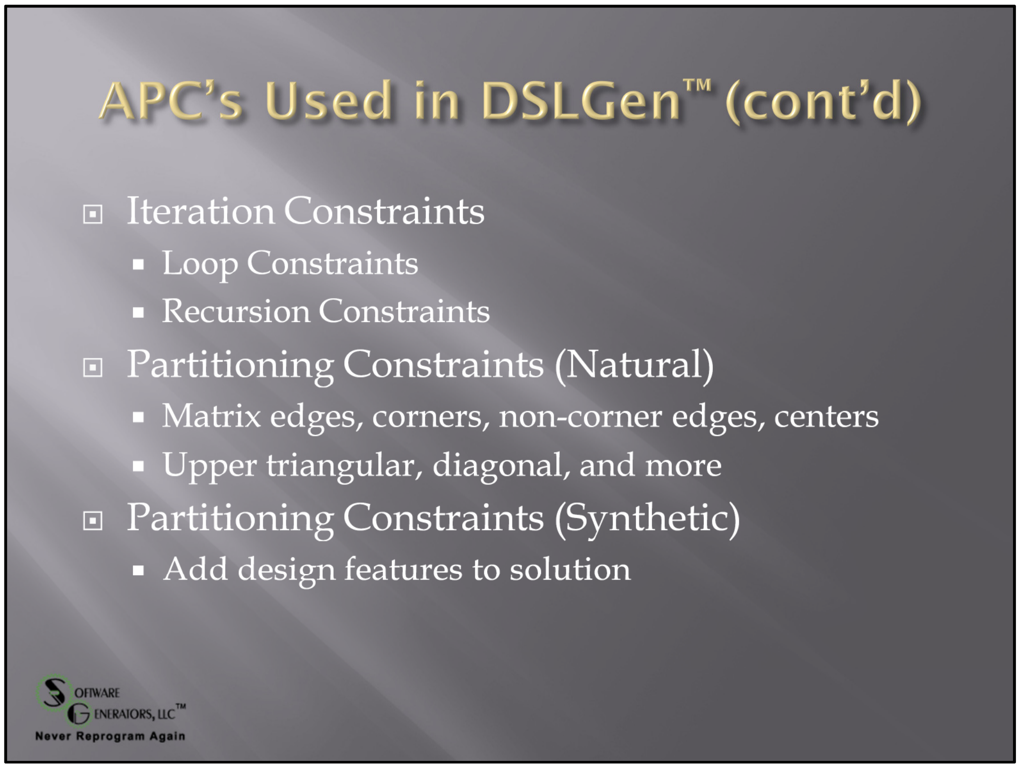
The DSLGen™ uses two broad categories of constraints (i.e., APCs) in defining the Logical Architecture of a domain specific computation: Iteration constraints and partitioning constraints. Iteration constraints are further divided into two subdivisions – loop and recursion constraints. Because of the nature of our domain examples (Digital Signal Processing), we will focus on loop constraints. A loop APC is the nominal description of one or more loops over some problem domain data structure (e.g., a grayscale digital image), meaning that it provides the essence of the computational iteration structure without making any commitment to how that iteration structure will be expressed in the target implementation program. Or put differently, it tells "what" the computation is computing without telling exactly "how" the implementation program will achieve that "what."

The second broad category of constraints (APCs) is the Partitioning constraint, which provides a provisional division of loop APCs (and computations in general) into parts, where these "parts" may be located and used in some as yet undetermined substructure of the target implementation program form. That as yet undetermined substructure will likely be provided by a design framework that provides 1) the broad architectural context and structure of the overall program implementation form of the computation, 2) some "holes" within that context designed to receive those aforementioned parts and 3) the interstitial (i.e., connective) tissue that ties the parts together with each other and with the design framework context.

Partitioning constraints are further divided into two types (i.e., Natural and Synthetic). These respectively distinguish between partitioning that is inherent to the specific computation being generated (i.e., Natural Partitions) or partitioning that is the result of elective design features that the user desires to have in the target program implementation (i.e., Synthetic Partitions).

Natural partitions are instances of problem domain specific types/classes within DSLGen™. These types/classes form a problem domain language by which to talk about (i.e., specify) parts of a computation without having to express the parts in terms of programming language abstractions and structures. So, in the DSP domain, one can express the concepts of image edges, centers, corners, non-corner edges, and so forth. This is an open-ended language that can be easily extended to include meaningful partitions for the DSP domain or any other domain of interest (e.g., data structures, user interface, network, web, IT, etc.). For example, the closely related domain of numerical analysis computations includes concepts like upper triangular, diagonal, singular matrix, among many others.

(continued on next slide)



APC's Used in DSLGen™ (cont'd)

- Iteration Constraints
  - Loop Constraints
  - Recursion Constraints
- Partitioning Constraints (Natural)
  - Matrix edges, corners, non-corner edges, centers
  - Upper triangular, diagonal, and more
- Partitioning Constraints (Synthetic)
  - Add design features to solution

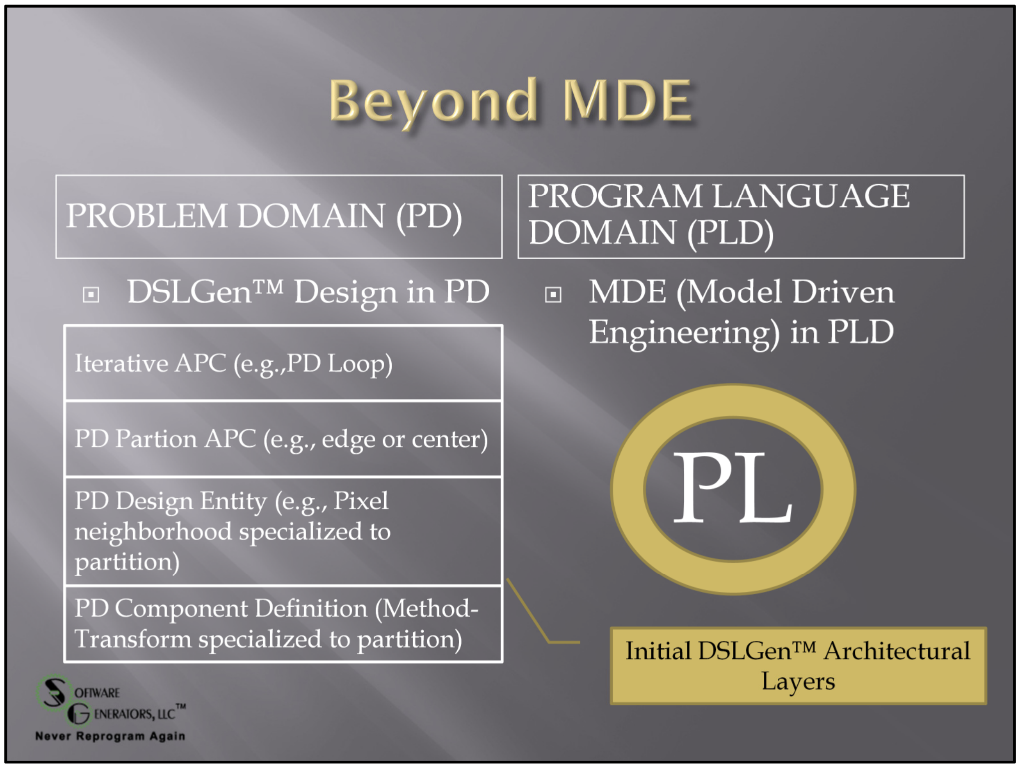
Software Generators, LLC™
Never Reprogram Again

(This hidden slide allows continuation of speaker's notes for this slide.)

Synthetic partitions provide the facility to introduce implementation domain concepts, that is, concepts that encapsulate elective design features that will relate to design frameworks and let us build design framework (DF) patterns that capture architectural structures in the DF without having to descend into the detail of and commitment to programming level structures and operations. In an example that we will look at, a synthetic partitioning concept will define a "slice of an image center", which will mean that it is a specialization of the natural partitioning constraint center and thereby, will inherit those properties of a center partition (e.g., definitions of programming components specific to a center such as a neighborhood weight function) that are no different from (and therefore, not specialized for) a slice of an image center. However, other properties that are specific to a slice of an image center (e.g., definitions that determine loop ranges for a slice of an image center) will be specialized for it. Thus, such component definitions are said to "encapsulate the design feature" of a slice of an image center and the code that is generated from those definitions will correctly express that design feature in the target program implementation code.

APCs allow DSLGen™ to build a functionally-based logical architecture (LA) of the computation and evolve that LA (via the introduction of synthetic partitions) to an architecture that meshes with some target design framework and thereby can be cast into actual code that has the desired properties. Functionally based definitions of components, like a neighborhood weight function for a convolution, provide the "referential transparency" property, which allows the definitions to be relocated within the LA (and to a limited degree within a DF) without the kind of dependencies that programming language abstractions would introduce. That is, "referential transparency" means that the generator does not have to deal with side-effects of the component definitions.

So, to get a tiny bit more concrete, the next slide shows an abstract view of an initial LA for a specific example.



Beyond MDE

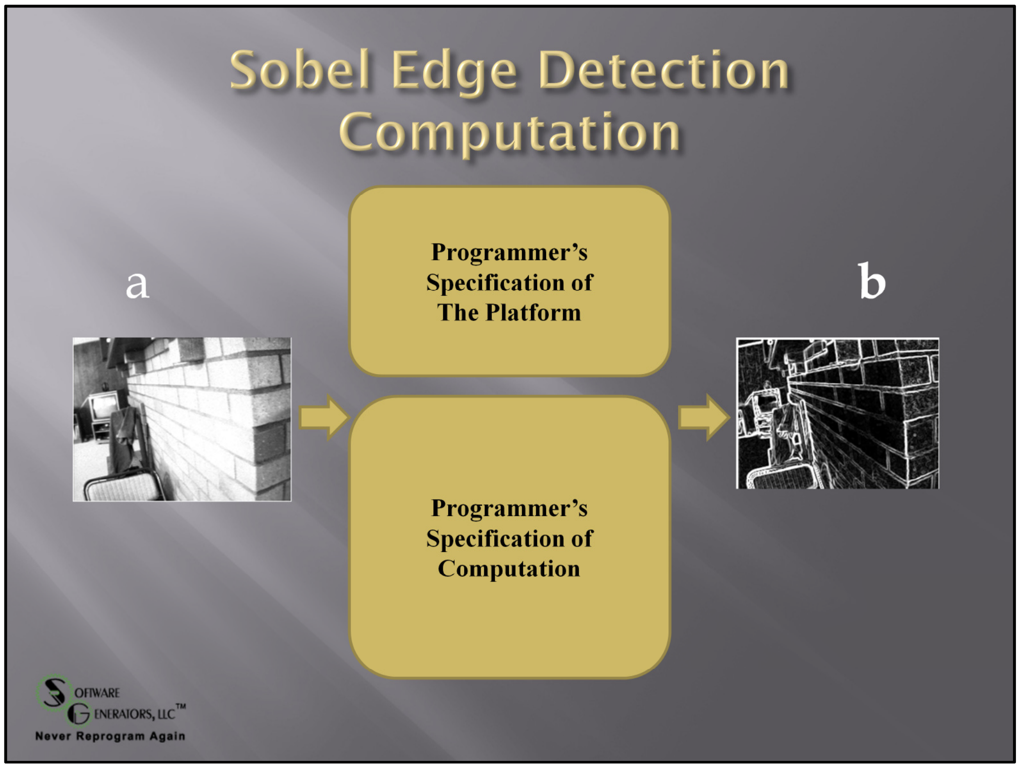
PROBLEM DOMAIN (PD)	PROGRAM LANGUAGE DOMAIN (PLD)

- DSLGen™ Design in PD
- MDE (Model Driven Engineering) in PLD

Iterative APC (e.g.,PD Loop)
PD Partion APC (e.g., edge or center)
PD Design Entity (e.g., Pixel neighborhood specialized to partition)
PD Component Definition (Method-Transform specialized to partition)
PL
Initial DSLGen™ Architectural Layers
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Here is an abstract example of a set of initial LA layers, where a problem domain loop (e.g., a loop over a grayscale image or RGB image) is by implication partitioned into edges and centers because of the partition APC that modifies it. Within an edge partition (e.g., the top edge of the image, just to get concrete), the neighborhood (e.g., sp) is specialized into a neighborhood specific to that partition (e.g., sp-edge1). Furthermore, that neighborhood specialization will trigger the component definitions [e.g., the Method-Transform (MT) defining (w sp …)] to be specialized to the specific partition (e.g., edge1) producing a Method-Transform like (W sp-edge1 …). To achieve this specialization, the generator assumes the partitioning condition of one of the partitions is true (e.g., (== i 0)), substitutes "t" for that expression and partially evaluates the RHS of the MT (w sp …). The result of that partial evaluation becomes the new RHS of the definition of (w sp-edge1 …). In a few moments, we will show how this specialization occurs (see the IL Specializations slide below).

Thus, each partition object has a set of component definitions specialized to it for building code elements that are specific to that definition. These specialized definitions "encapsulate" the design feature of "edge-ness" or "center-ness" of digital images. When the generator in-lines these definitions, we can be assured that they will generate code that reflects this design feature. A little later in the talk, we will see some concrete examples of LA-s as seen by a domain engineer who is using a tool called the Architecture Browser in the course of his debugging or extending the generator.



Sobel Edge Detection Computation

a

b

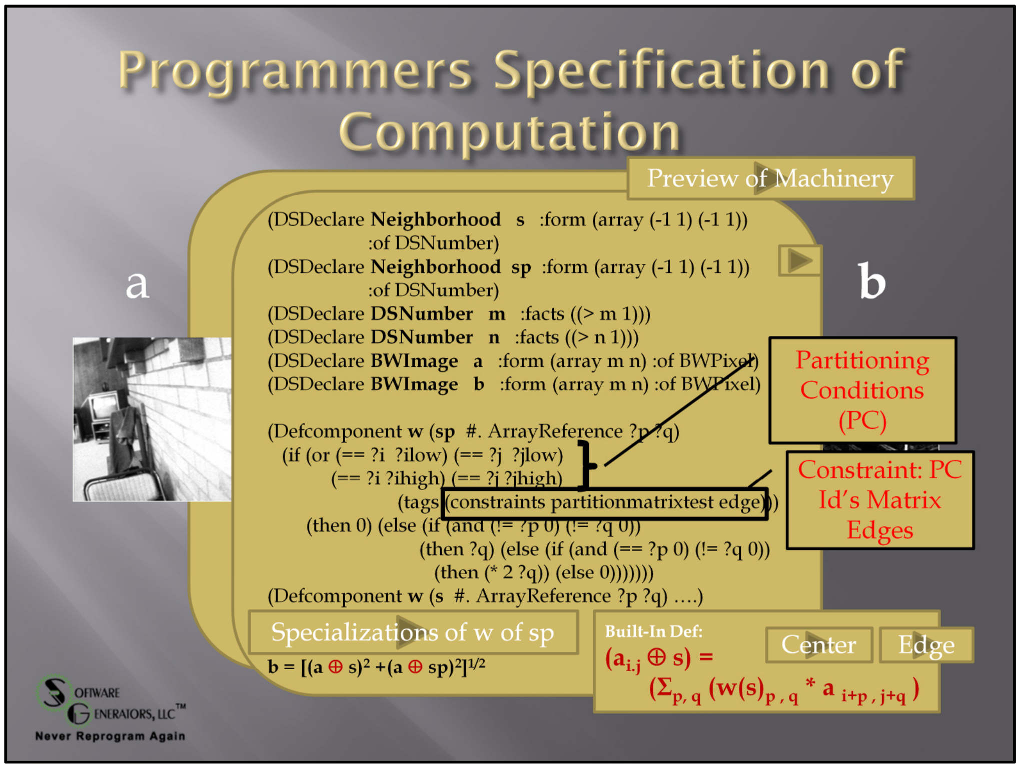
Programmer's Specification of The Platform

Programmer's Specification of Computation

OFTWARE ENERATORS, LLC™
Never Reprogram Again

So, let's look at the two specifications provided by the application programmer to DSLGen™ to generate differing implementation code for differing execution platforms. We will start with the specification of the computation of Sobel edge detection. **(Reader's Note: the following slide is a "build" slide and the speaker's notes for that slide reflect this. If you are reading this presentation as a pdf file, you will have to use your imagination!)**

(NB: the word "edge" is an overloaded term in the context of this talk. The Sobel algorithm is referring to visual edges in an image and "edge" in DSLGen™ is a subclass of the matrixpartition class (which itself is a subclass of the APC class). Thus, that latter "edge" is a partition. These kinds of word collisions are unavoidable in a semantically rich environment like this but I assume that the listener can easily disambiguate based on context.)



Programmers Specification of Computation

Preview of Machinery

a

b

```
(DSDeclare Neighborhood  s   :form (array (-1 1) (-1 1))
                :of DSNumber)
(DSDeclare Neighborhood  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)

(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)))))))
(Defcomponent w (s  #. ArrayReference ?p ?q) ….)
```

Partitioning Conditions (PC)

Constraint: PC Id’s Matrix Edges

Specializations of w of sp

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

Built-In Def:

$(a_{i,j} \oplus s) = (\Sigma_{p,q}\, (w(s)_{p,q} * a_{i+p,\, j+q}\,)$

Center

Edge

SOFTWARE GENERATORS, LLC™

Never Reprogram Again

First mouse click – the heart of Sobel edge detection expressed in the Image Algebra (IA) as the formula that we have seen and analyzed earlier. **$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2}$**

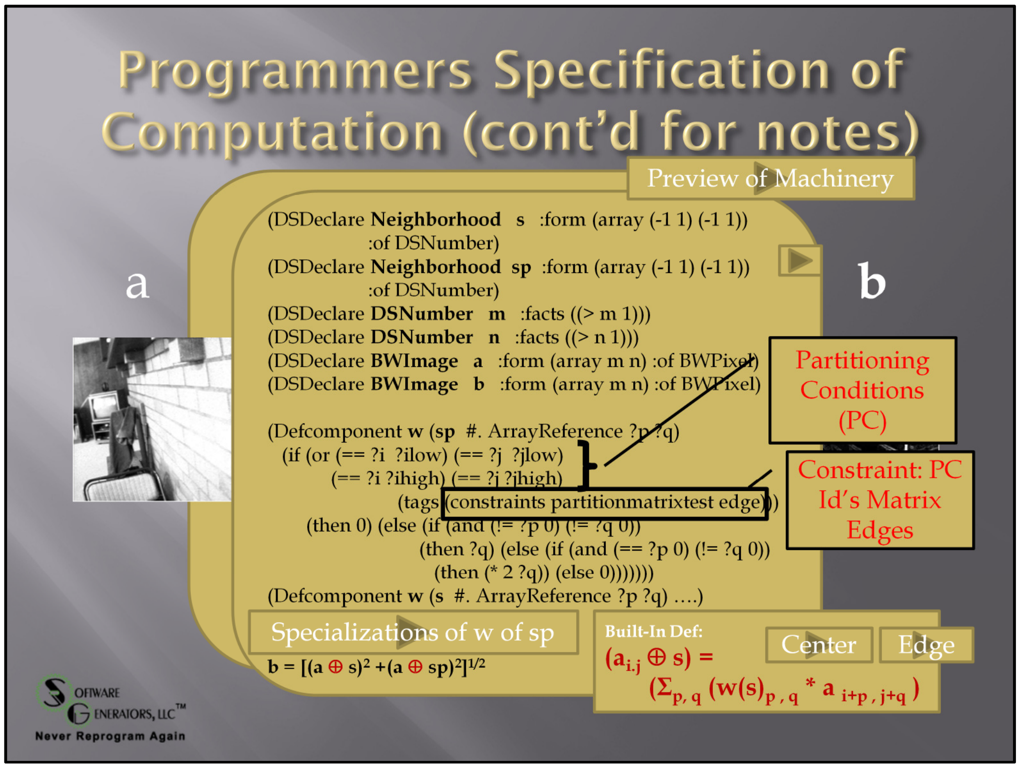
Now we need some definitions for the elements of that expression. Let’s start with the two images, a and b. In DLSGen™ these are expressed as declarations that look like … **second mouse click** …

These definitions for images a and b use the built-in type BWImage. They further declare that a and b are M by N matrices (i.e., “**:form (array m n)**”) containing BWPixels (i.e., “**:of BWPixels**” indicates grayscale pixels). BWPixels will eventually get mapped to a C language type, e.g., int.

We also have to declare m and n as the built-in type DSNumber and again, this type will eventually get mapped to a C type like int. Notice there are no initial values for m or n. The application programmer will have to provide them or provide code that provides them.

Notice also that there is a logical expression in the “**:facts**” slot of m and n restricting those values to be greater than 1. Why? Any object in DSLGen™, and virtually every data entity the generator works with, is a CLOS object and will have a “facts” slot that may contain a conjunctive list of assertions that provide facts about that object. These facts may be used by the partial evaluator (PE) when it is trying to simplify a logical expression like “(i == (m + 1))” in a partition context in which it is already known that “(i == 0)”. The partial evaluator will call the simultaneous inequality equation solver to try to determine whether “(i == (m + 1))” is true, false or unknown. Given, the fact that (m>1), the solver will be able to prove that “(i == (m + 1))” is false when “(i == 0)”. Based on this result, the PE can simplify “(i == (m + 1))” to false, which may well lead to more extensive code simplifications (e.g., if “(i == (m + 1))” is the test condition of an if-then-else statement, that if-then-else would simplify to just the else clause).

Third mouse click – Now, we need declarations for the neighborhoods s and sp. The Neighborhood type is another built-in definition. The s and sp neighborhoods are declared to be 2 by 2 arrays that, in the Image Algebra domain specific language, are indexed from -1 to 1 on both dimensions. This is to allow the center pixel of the neighborhood to be at [0,0] of the neighborhood. Of course, when mapped to C, these ranges will become 0 to 2.



Programmers Specification of Computation (cont’d for notes)

Preview of Machinery

a

b

```
(DSDeclare Neighborhood  s   :form (array (-1 1) (-1 1))
                :of DSNumber)
(DSDeclare Neighborhood  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)

(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)))))))
(Defcomponent w (s  #. ArrayReference ?p ?q) ….)
```

Partitioning Conditions (PC)

Constraint: PC Id’s Matrix Edges

Specializations of w of sp

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

Built-In Def:

$(a_{i,j} \oplus s) = (\Sigma_{p,q} (w(s)_{p,q} * a_{i+p,j+q})$

Center Edge

SOFTWARE GENERATORS, LLC™ Never Reprogram Again

(This hidden slide is included to allow continuation of the speaker’s notes.)

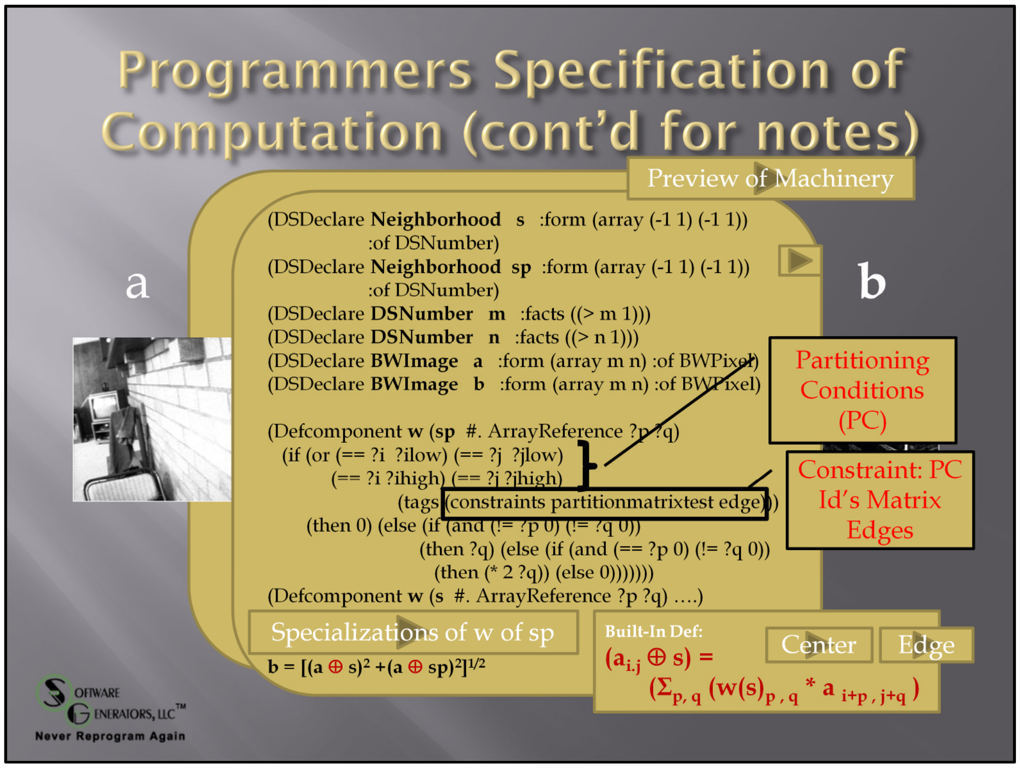
Now, we need some definitions for the domain specific operators. Let’s start with convolution, which is built-in. In fact, the IA provides a number of kinds of convolutions (e.g., sum of products, product of sums, max of all pair max-s, min of all pair min-s, and exclusive or). But for this example, we will need the sum of products version of convolution, which is defined as **Fourth mouse click.**

This definition suggests that the computation walks over the neighborhood summing up the products of a pixel and a neighborhood weight defined for each [p+offsetp,q+offsetq] position within the neighborhood.

(**Speakers NOTE:** IF needed, the **Center Action button** will jump to the Intuitive computational essence of convolution for an a[i,j] pixel in the center part of the image followed by an analogous slide for an edge pixel. The **Return Action Button** on the edge pixel essence slide will return directly to this slide. **Return Action Buttons return to the last slide viewed, so follow the sequence precisely or risk getting lost in the presentation sequence.**)

Now, we need definitions for (w sp ….) and (w s …). **Fifth and sixth mouse click.** The definitions of w of sp and w of s are so-called **method-transforms** (**MTs**) because internally they become transformations that look and largely behave like methods. “w” behaves like a method name. sp and s behave somewhat like classes in that the w definitions are inherited up the type hierarchy starting with sp and s. For example, if we have a specialization of sp, say sp-edge1 but there is no MT that matches (w sp-edge1 …), then the MT defined for (w sp ….) would match and that transform would fire. Note that transforms are enabled only for specific NAMED generator phases and MT’s are enabled (by implication) only for the phase that does inlining of definitions (i.e., the **“formals**” phase) **,** which is rather late in the generation process after the code design process is completed.

So, in the definition of w of sp, what looks like a parameter sequence is actually a pattern specification, which will try to match some AST subtree and will succeed if that subtree is of the form “(w sp ….)” or, as noted earlier, some other more specialized form based on a specialization of sp (e.g., sp-edge1). I don’t want to get too deeply into the weeds with this pattern matching process because the details are not really relevant for the level that we are describing. Suffice it to say, that pattern building parts (like ArrayReference) are defined elsewhere and do a lot of fetching of relevant information and binding that information to the pattern variables (e.g., ?i, ?j, ?ihigh, etc.). This pattern will get the names of the indexes traversing the image (e.g., “i” and “j”), their ranges (e.g., ? ilow and ?ihigh, which might be bound to “0” and “(- m 1)”), and the indexes traversing the neighborhood (e.g., ?p and ?q) among other data.



Programmers Specification of Computation (cont'd for notes)

Preview of Machinery

a

b

```
(DSDeclare Neighborhood  s  :form (array (-1 1) (-1 1))
               :of DSNumber)
(DSDeclare Neighborhood  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)

(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)))))))
(Defcomponent w (s  #. ArrayReference ?p ?q) ....)
```

Partitioning Conditions (PC)

Constraint: PC Id's Matrix Edges

Specializations of w of sp

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

Built-In Def:

$(a_{i,j} \oplus s) = (\Sigma_{p,q} (w(s)_{p,q} * a_{i+p,j+q})$

Center

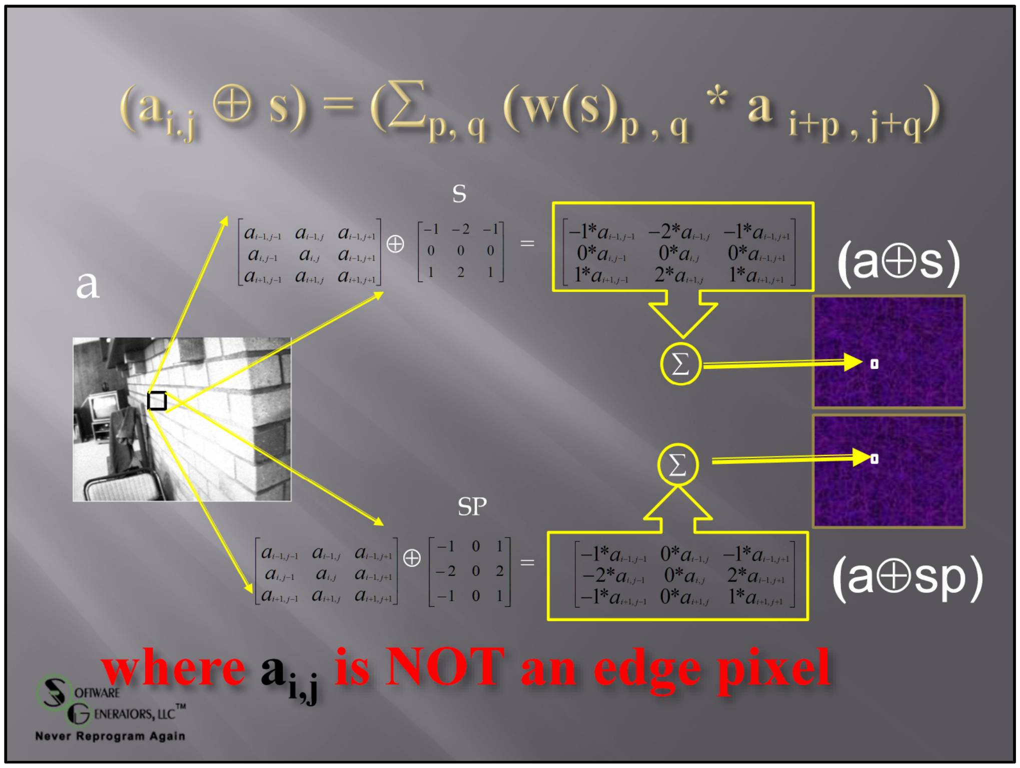
Edge

(This hidden slide is included to allow continuation of the speaker's notes.)

The RHS of the transform is the functional expression that is the body of the MT definition. That body basically says if either ?i or ?j is an edge pixel, the definition is 0. Otherwise, it is a messy expression that, if one walks through it on paper, one will derive the matrix of weights for sp shown in convolution essence diagram for NON-edge pixels. **(Optionally, the speaker can use the Center Action Button to traverse to the Essence slide for center to show the values. But be sure to then click to next slide (edge pixels) and finally click the Return Action Button to come back HERE to further explain the key idea of Partitioning Conditions.)**

Now, a **key idea** of this definition is that the application programmer is providing DSLGen™ with domain specific knowledge about the natural partitioning of this computation. He or she is doing this by the property list (a so-called "tags" list) that provides information about the (or …) expression. **Seventh mouse click.** The tags list provides a "constraints" property that defines the (or …) expression as a partitioning condition for this computation, identifies the kind of partitioning as "partitionmatrixtest" and identifies the kind of domain object(s) that the partitioning condition identifies (i.e., matrix edges). This will trigger construction of a set of four edge partition objects (one for each disjunct) and an implied center partition for the negative case. The construction of the partitioning objects further triggers the creation of specialized definitions of MTs for each partition object so that when code is generated for expressions within the contexts of these different computational partitions, the definition specific to that computational context (i.e., specific to that computational partition) will be used to generate that code. We can see an example of how these specialize definitions are derived by showing an example for this MT definition. **(Speaker's note: Click on the "Specializations of w of sp" Action Button. This goes to a build chart requiring 5 clicks to fully explain. But sure to return HERE using the Return Action Button upon completion of the slide.)**

Now, we have all of the computational definitions we need to follow through a Sobel example. So now, let's examine what the execution platform specification looks like. **Click the Go To Preview of Machinery Action button to leave this slide for good.**



$$(a_{i,j} \oplus s) = (\sum_{p,q} (w(s))_{p,q} * a_{i+p,j+q})$$

$$\begin{bmatrix} a_{i-1,j-1} & a_{i-1,j} & a_{i-1,j+1} \\ a_{i,j-1} & a_{i,j} & a_{i-1,j+1} \\ a_{i+1,j-1} & a_{i+1,j} & a_{i+1,j+1} \end{bmatrix} \oplus \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = \boxed{\begin{bmatrix} -1*a_{i-1,j-1} & -2*a_{i-1,j} & -1*a_{i-1,j+1} \\ 0*a_{i,j-1} & 0*a_{i,j} & 0*a_{i-1,j+1} \\ 1*a_{i+1,j-1} & 2*a_{i+1,j} & 1*a_{i+1,j+1} \end{bmatrix}}$$

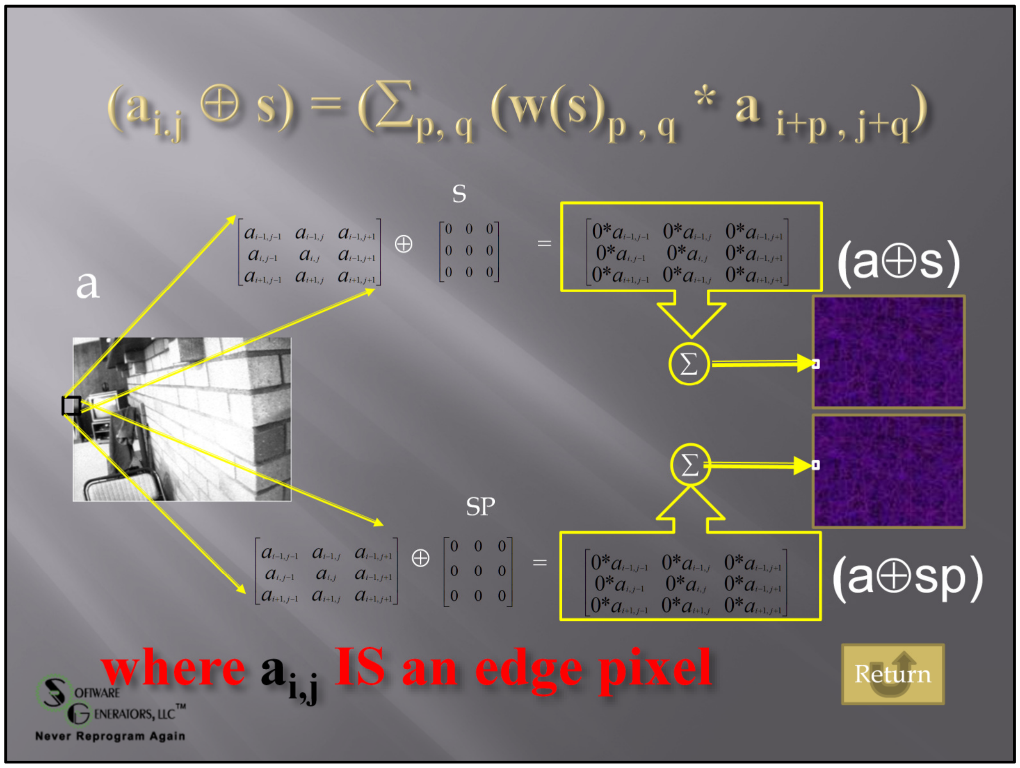
$$\Sigma$$

$$\begin{bmatrix} a_{i-1,j-1} & a_{i-1,j} & a_{i-1,j+1} \\ a_{i,j-1} & a_{i,j} & a_{i-1,j+1} \\ a_{i+1,j-1} & a_{i+1,j} & a_{i+1,j+1} \end{bmatrix} \oplus \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \boxed{\begin{bmatrix} -1*a_{i-1,j-1} & 0*a_{i-1,j} & -1*a_{i-1,j+1} \\ -2*a_{i,j-1} & 0*a_{i,j} & 2*a_{i-1,j+1} \\ -1*a_{i+1,j-1} & 0*a_{i+1,j} & 1*a_{i+1,j+1} \end{bmatrix}}$$

$$\Sigma$$

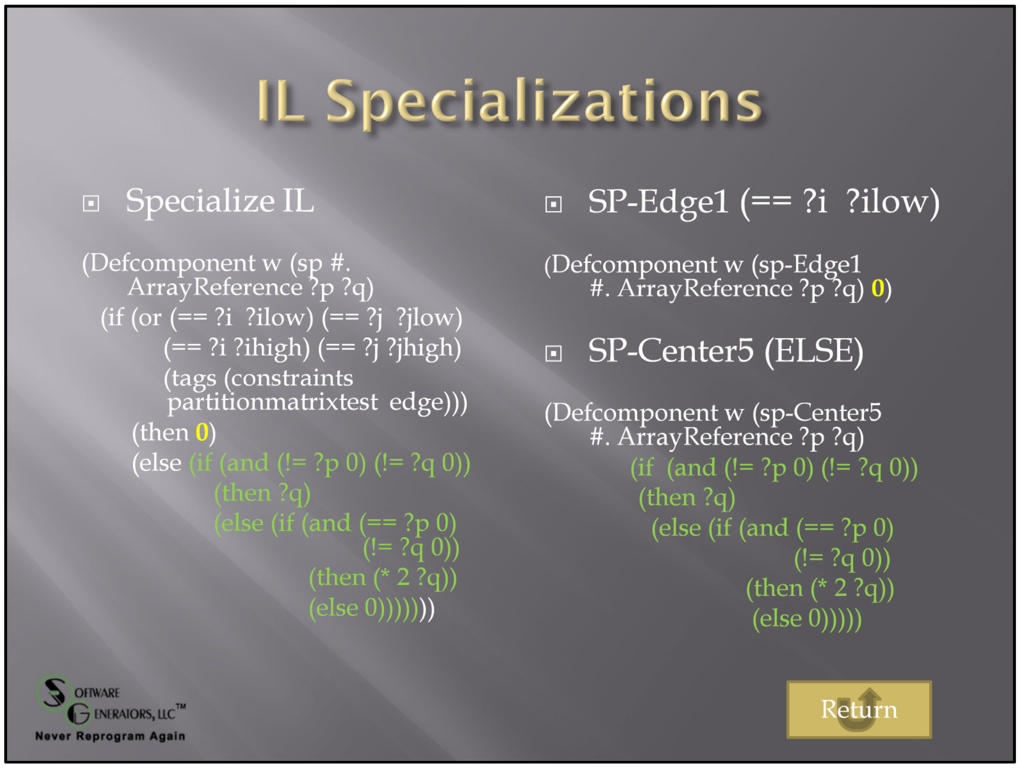
where $a_{i,j}$ is NOT an edge pixel

A convolution is computed for all non-edge pixels a[i,j] by pair wise multiplying a[i,j] and its neighboring pixels with the weights of the neighborhood s and then summing all of those values produces a pixel of the intermediate image (a convolve s). The neighborhood sp produces another image. Notice that the weights of s are the weights of sp rotated 90 degrees clockwise. Remember these operations just produce the non-edge portions of the first two intermediate images of the overall Sobel edge detection formula.


$(a_{i,j} \oplus s) = (\sum_{p,q} (w(s)_{p,q} * a_{i+p,j+q})$
a
S
SP
$(a \oplus s)$
$(a \oplus sp)$
where $a_{i,j}$ IS an edge pixel
Return
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

This is analogous to the center computation except that all weights of s and sp are 0 for edge pixels. Thus, the pixel value of (a convolve s) and (a convolve sp) will both be 0.

You may notice that we are being a bit sloppy with this definition. Why? Since a[i,j] is an edge pixel, at least one row or column of neighboring pixels will not exist (a column and a row for corner pixels). The actual definitions we will use will account for this but in Sobel Edge detection, this is masked by the fact that (w s …) and (w sp …) are zero for edges. So, even if we did not properly account for this, partial evaluation would eliminate the error. However, it does matter in other computations, e.g., image averaging. In image averaging, the neighborhood loops have varying beginning and ending indexes and these will manifest themselves in the implementation code generated by a conditional expression for the indexes if the computation is partitioned and by separated instances of neighborhood loops for the partitions (i.e., corners, non-corner-edges and centers).



IL Specializations

- Specialize IL

```
(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)))))))
```

- SP-Edge1 (== ?i ?ilow)

```
(Defcomponent w (sp-Edge1
    #. ArrayReference ?p ?q) 0)
```

- SP-Center5 (ELSE)

```
(Defcomponent w (sp-Center5
    #. ArrayReference ?p ?q)
       (if  (and (!= ?p 0) (!= ?q 0))
        (then ?q)
         (else (if (and (== ?p 0)
                        (!= ?q 0))
                   (then (* 2 ?q))
                   (else 0)))))
```

First Click – So, the objective is to produce new MT definitions that are specialized to the computational context of each different partition. We will use the MT defining the w function of the neighborhood sp as a concrete example. The procedure is to assume one partitioning condition is true and all others are false.

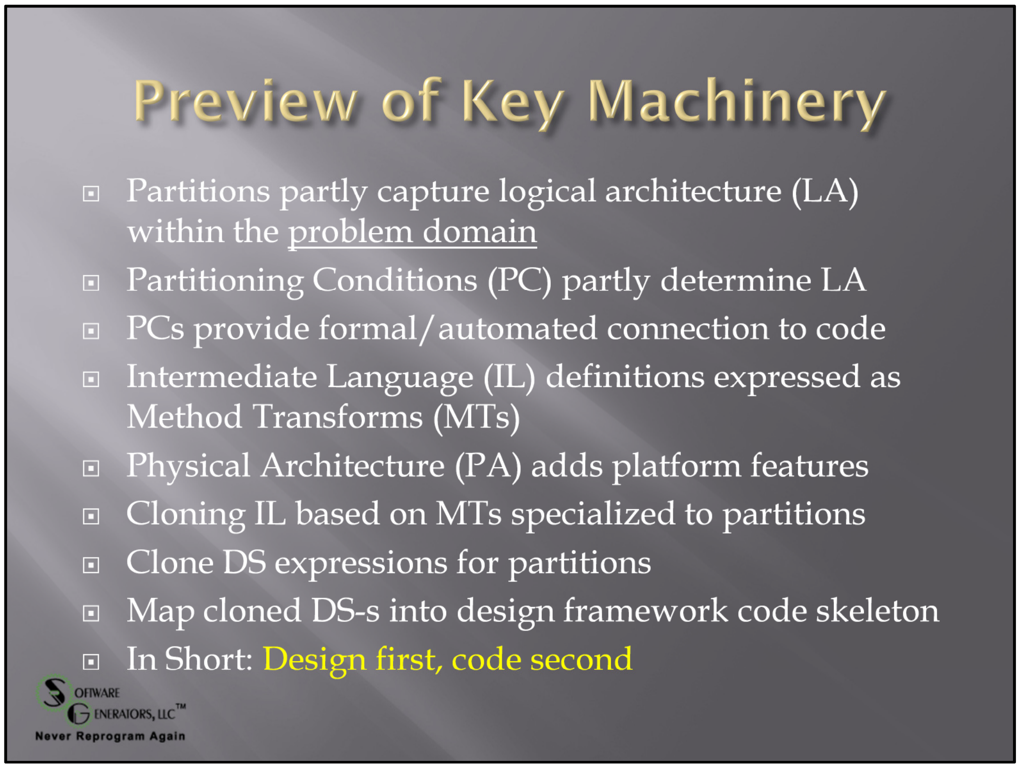
Second Click - We will start with the partitioning condition specific to the sp-edge1 neighborhood. The procedure substitutes t for the partitioning condition expression "(== ?i ?ilow)" and false (i.e., nil) in the other PC-s (although that is irrelevant in this example). Then DSLGen™ partially evaluates the RHS of the MT definition **(third click)** which produces a new MT definition that is specialized to sp-edge1.

Once all PC-s have used, the procedure processes the all-false case **(fourth click)**, i.e., for this example, the case that is specific to a center pixel in the image. Partially evaluating that set of substitutions will produce **(fifth click)** the a new MT definition specific to sp-center5 in this example.

There are a number of MTs that taken together provide the "Intermediate Language" used to stand-in for the real code during the design phase while the architecture of the overall code design is being built, scopes developed and design features being encapsulated into the design. The IL provides the ability to use the most general form of operators, loops, etc. during the design process but when specific partitions are finally mapped onto portions of the physical code design, those portions of the physical design will be automatically specialized to the correct code forms when the IL is inlined.

The overall set of IL MTs for the domain of convolution in digital signal processing include, IL for mapping from neighborhood coordinates to image coordinates, IL for the start, end and increment for various kinds of loops (e.g., image loops or neighborhood loops), IL for mapping to the partitioning condition expression for the specific partition being inlined and so forth. Other problem domains will require other IL subsets.

(If you arrived at this slide via an Action Button and need to return to that slide, i.e., return to the Computation Specification slide, then click the Return Action Button now.)



Preview of Key Machinery

- Partitions partly capture logical architecture (LA) within the problem domain
- Partitioning Conditions (PC) partly determine LA
- PCs provide formal/automated connection to code
- Intermediate Language (IL) definitions expressed as Method Transforms (MTs)
- Physical Architecture (PA) adds platform features
- Cloning IL based on MTs specialized to partitions
- Clone DS expressions for partitions
- Map cloned DS-s into design framework code skeleton
- In Short: Design first, code second

Software Generators, LLC™
Never Reprogram Again

Partition based logical architecture (LA) of the computation:

- Which models the natural case structure of the computation, where the case structure is determined by some set of partitioning conditions (PC)
- Which exists within the problem domain not the programming language domain,
- Whose organization is only one of potentially many APC constraints on the final organization of the generated code (i.e., organization within the programming language or implementation domain)
- Whose application (to formulate the actual code) is deferred, and
- Where the LA can be changed, extended and evolved before it is actually applied to formulate code.

An intermediate language (IL) for expressing generalized forms of operator definitions (e.g., ⊕, w, row, col, etc.)

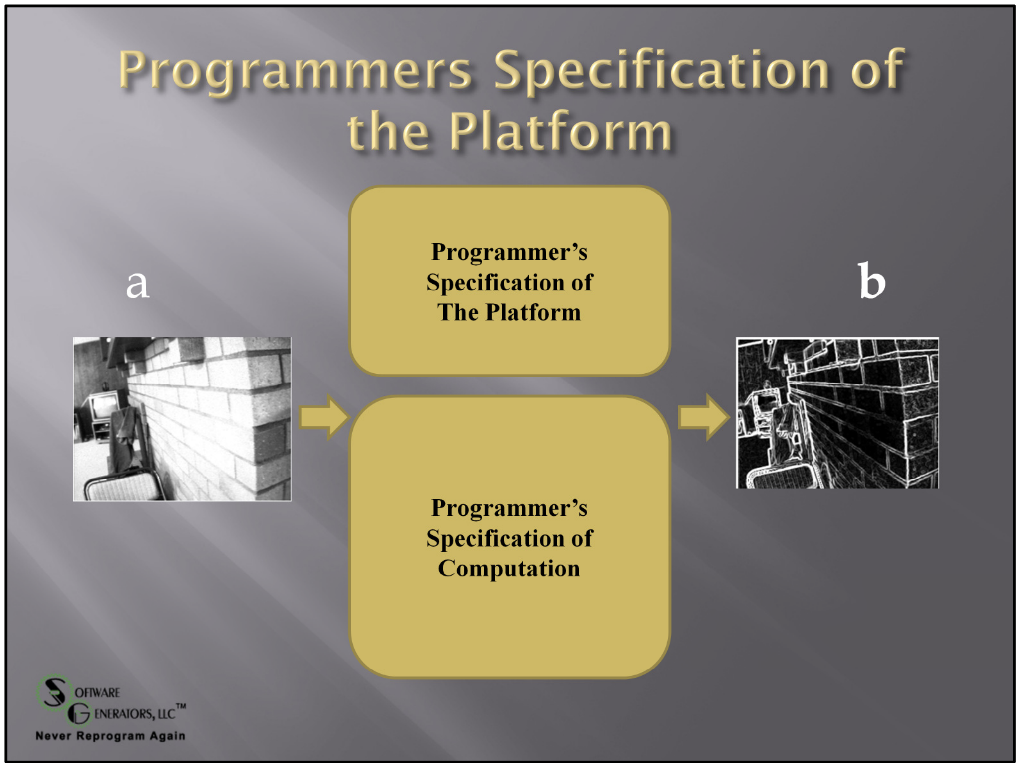
- Where transformations that look and behave like methods (i.e., "MTs") serve as generic definition mechanisms (e.g., recall (w sp ….) definition),
- Where Domain specific operator (e.g., ⊕ convolution) definitions initially can be defined in **generic** IL terms (e.g., generic ⊕ will be defined in terms of the IL MTs w, col, row)
- Where the IL definitions may be specialized to capture coding differences specific to different partitions (e.g., recall specialization of (w sp …) to (w sp-edge …) or (w sp-center …)),
- Where differences may be determined by partition specific design entities (e.g., pixel neighborhood entities like sp-edge and sp-center),
- Where specialized (i.e., partition dependent) IL definitions are associated with partition specific locales of the Logical Architecture structure
- Where cloning and specialization of the generic operator definitions to specific partitions can be deferred until partitions are ready to be mapped into the final code organization.

All of this machinery is designed to allow DSLGen™

- To build logical design architectures (LAs) largely within the problem domain at first,
- To evolve those LAs to physical architectures (PAs) (a.k.a. "synthetic architectures" or SAs) by adding implementation/execution platform specific design features next,
- To clone the domain specific expression of the computation (associated with some partitioned LA) for the different partitions of that LA, and then
- To map the cloned DS expressions into "holes" within design framework code skeletons.

In short, design first within the problem domain, code second within the programming domain (a.k.a. the execution platform domain).

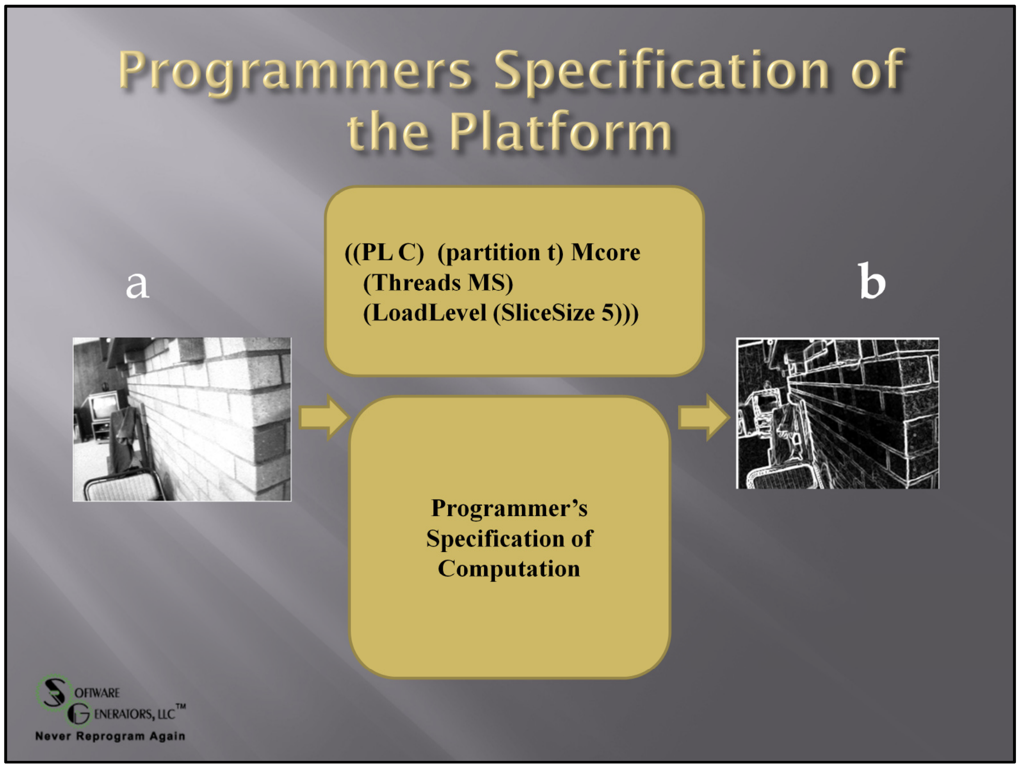
Now, let's look at this machinery in action via a concrete example.



Programmers Specification of the Platform

a
b
Programmer's Specification of The Platform
Programmer's Specification of Computation
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Now that we have a sense of how to specify the target computation that we want to generate implementation code for, let's re-examine the specification of the execution platform for thread based parallelism on a multi-core machine.



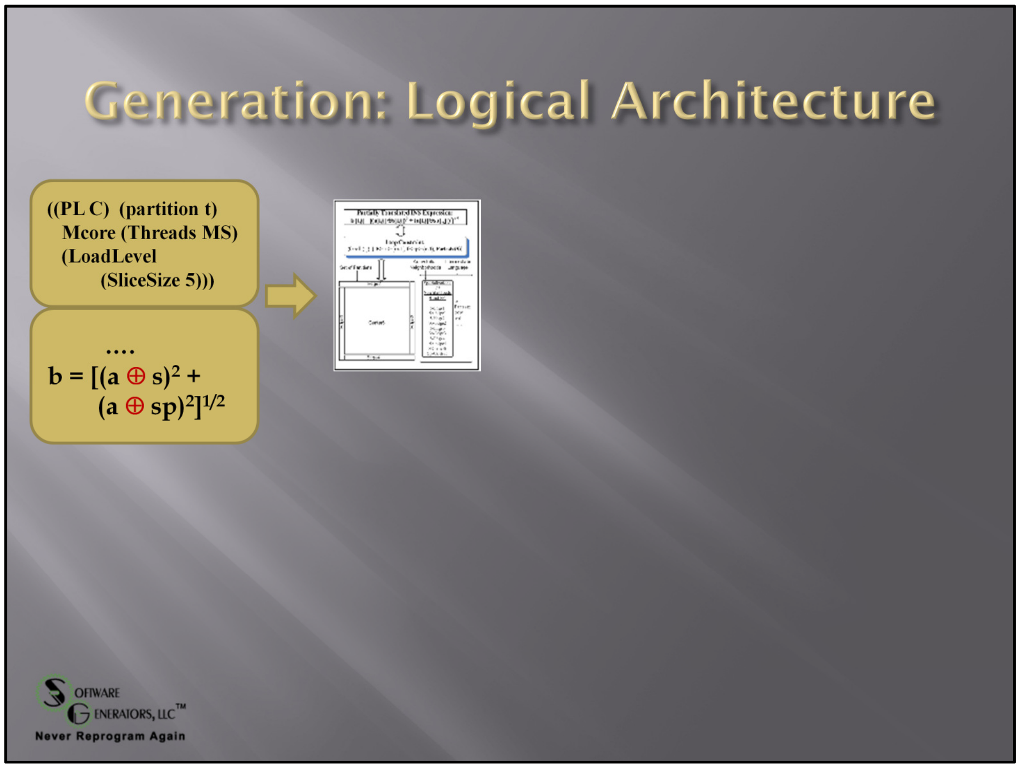
Programmers Specification of the Platform

```
((PL C)  (partition t) Mcore
   (Threads MS)
   (LoadLevel (SliceSize 5)))
```

a
b
Programmer's Specification of Computation
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Recall that we have seen this execution platform spec earlier in this presentation. To recapitulate, this spec is asking for C code with partitioning, optimization for multicore with MS threads and load leveling for heavyweight computations.

Within the generation process that follows, we will see the domain knowledge that an image center computation is a "heavyweight computation" will be brought to bear.



Generation: Logical Architecture

((PL C) (partition t)
  Mcore (Threads MS)
  (LoadLevel
        (SliceSize 5)))

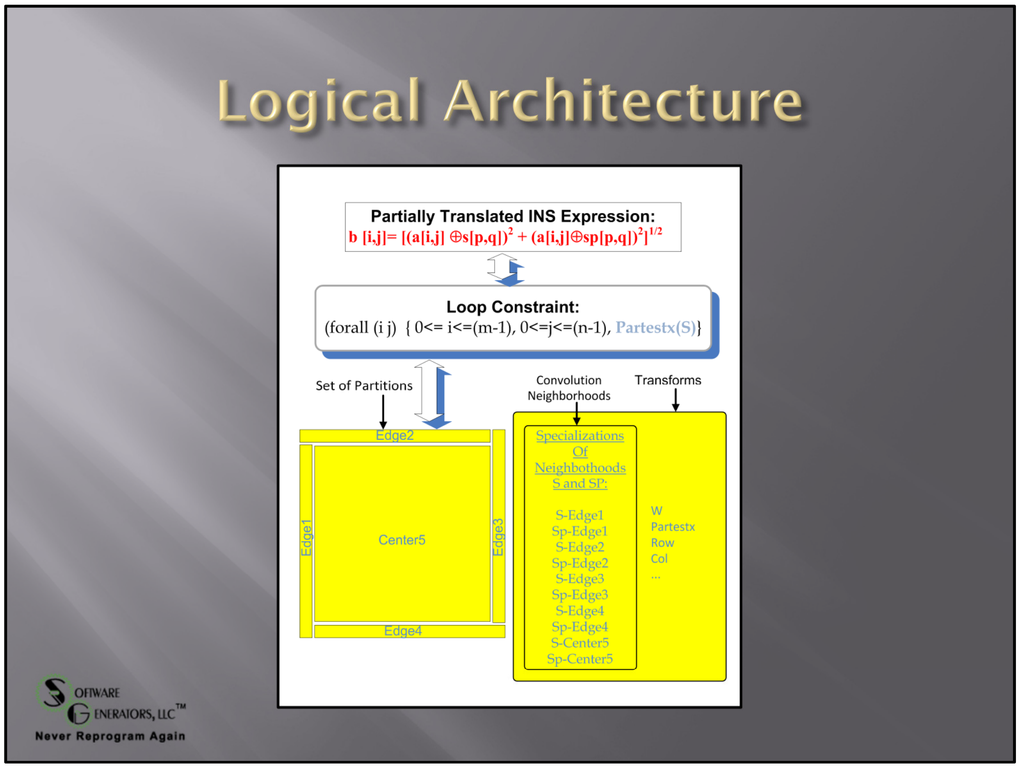
….

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

OFTWARE ENERATORS, LLC™
Never Reprogram Again

So, the first major generation phase partially translates the domain specific specification of the computation from images to pixels by introducing loop constraints, which in turn triggers generation of a provisional, logical architecture (LA) for the target code. Conceptually, this LA looks like --

Logical Architecture


Partially Translated INS Expression:
b [i,j]= [(a[i,j] ⊕s[p,q])² + (a[i,j]⊕sp[p,q])²]^1/2
Loop Constraint:
(forall (i j) { 0<= i<=(m-1), 0<=j<=(n-1), Partestx(S)}
Set of Partitions
Convolution Neighborhoods
Transforms
Edge2
Edge1
Center5
Edge3
Edge4
Specializations Of Neighbothoods S and SP:
S-Edge1
Sp-Edge1
S-Edge2
Sp-Edge2
S-Edge3
Sp-Edge3
S-Edge4
Sp-Edge4
S-Center5
Sp-Center5
W
Partestx
Row
Col
…
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

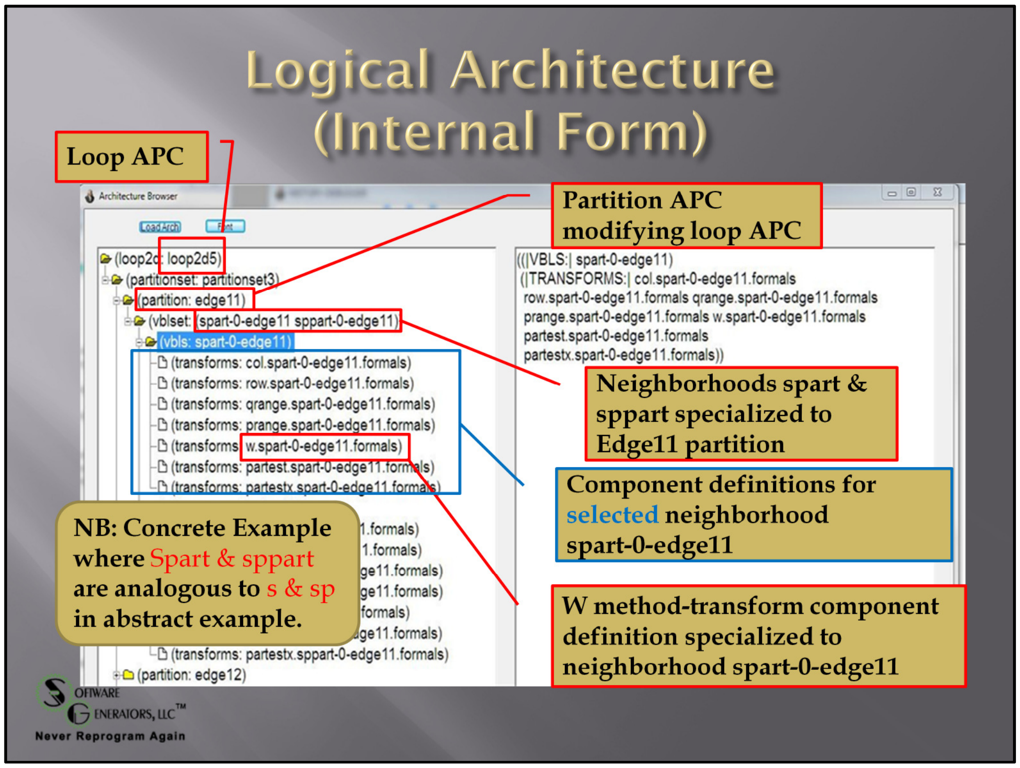
This phase translates the INS from images to pixels by introducing a 2D (two-dimensional) loop constraint and inventing provisional names "i" and "j" for the loop constraint's indexes, which will traverse the images a and b. This loop constraint contains (among other provisional information) the ranges and increment sizes of i and j, as well as "Partestx(S)", which is Intermediate Language (IL) that is the generic representation of the partitioning condition for the loop. This intermediate language (IL), in the context of some specific partition, will resolve to the concrete partitioning condition for that specific partition (e.g., it might resolve to "(i==0)). Thus, the IL (at this stage of the loop constraint's evolution) has not yet committed the loop to a specific context. A specific context will cause the loop to take on different forms for different contexts. For example, Partestx(S) for a neighborhood within an edge partition (e.g., S-Edge1) may (and in fact in this example will) cause the loop to have a different form than a loop with a Partestx for a neighborhood of an center partition (e.g., S-Center5).

The "fat arrow" indicates that the loop constraint is "associated" with the partially translated domain specific spec of the computation (i.e., the INS), which means that the loop constraint acts like a modifier to the meaning of the INS expression.

In the course of this part of the generation process, five partition constraints are also created (based on the tags list from the application programmer in the definitions of W of the neighborhoods sp and s). Actually, each definition of w will generate its own set of five partitions (edges1 thru 4 and center5, and edges6 thru 9 and center10) but as these propagate up the INS expression tree, the generator will discover that the partitions of the two sets have identical partitioning conditions and therefore, the two sets will be merged into (edges11 thru 14 and center15). (Note, there is an animation walk thru of this overall LA generation process shown in the supplemental Tools Demo presentation, also provided on the Software Generators web site.)

In the course of this process, specializations of the neighborhoods s and sp are generated and their Method-Transforms (MTs) that comprise the IL are specialized as we illustrated earlier. Some MTs may not require explicit specialization (i.e., they don't vary with respect to the partitioning condition). In that case, no explicit specialized versions need be generated and inheritance will produce the correct code during the inlining phase (which happens later). That is, if there is no MT definition for the exact AST expression (row s-edge1 ….), then inheritance will cause the MT that matches the AST expression (row s …) to trigger instead and produce the correct result.

Now, let's look at an actual example of a Logical Architecture as would be seen by a domain engineer who might be debugging or extending the generator's domains.



Logical Architecture (Internal Form)

Loop APC

Partition APC modifying loop APC

Neighborhoods spart & sppart specialized to Edge11 partition

Component definitions for selected neighborhood spart-0-edge11

W method-transform component definition specialized to neighborhood spart-0-edge11

NB: Concrete Example where Spart & sppart are analogous to s & sp in abstract example.

```
(loop2d: loop2d5)
  (partitionset: partitionset3)
    (partition: edge11)
      (vblset: (spart-0-edge11 sppart-0-edge11))
        (vbls: spart-0-edge11)
          (transforms: col.spart-0-edge11.formals)
          (transforms: row.spart-0-edge11.formals)
          (transforms: qrange.spart-0-edge11.formals)
          (transforms: prange.spart-0-edge11.formals)
          (transforms: w.spart-0-edge11.formals)
          (transforms: partest.spart-0-edge11.formals)
          (transforms: partestx.spart-0-edge11.formals)
                      1.formals)
                      1.formals)
                      ge11.formals)
                      ge11.formals)
                      formals)
                      ge11.formals)
          (transforms: partestx.sppart-0-edge11.formals)
    (partition: edge12)
```

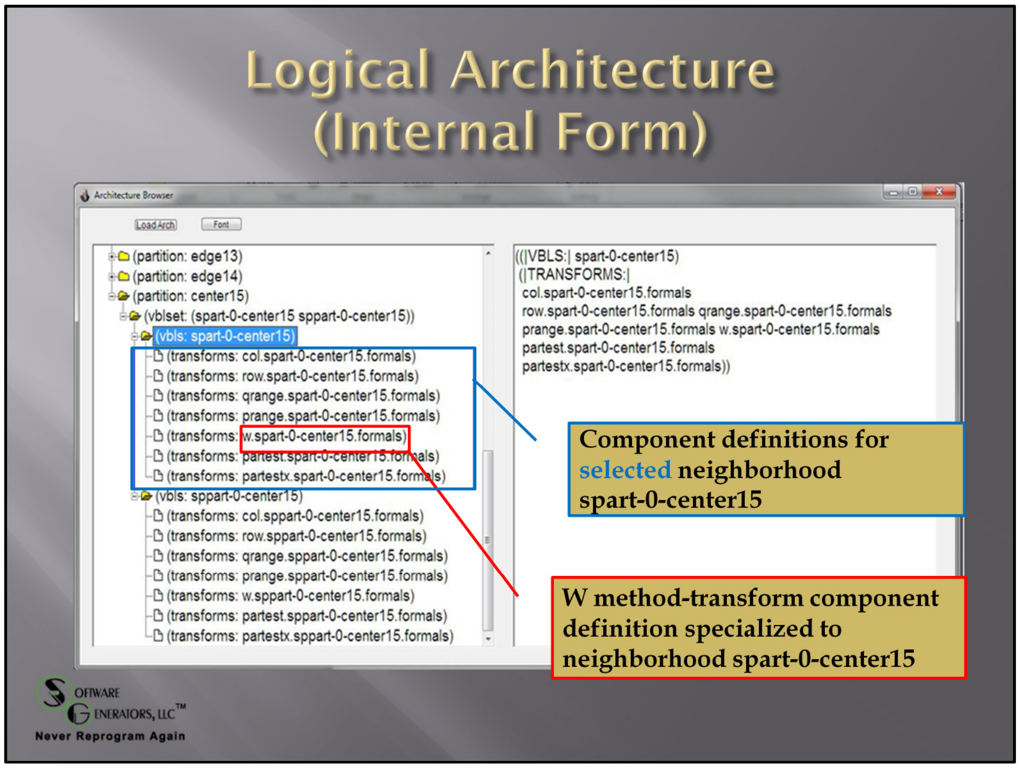
```
((|VBLS:| spart-0-edge11)
 (|TRANSFORMS:| col.spart-0-edge11.formals
  row.spart-0-edge11.formals qrange.spart-0-edge11.formals
  prange.spart-0-edge11.formals w.spart-0-edge11.formals
  partest.spart-0-edge11.formals
  partestx.spart-0-edge11.formals))
```

This is an actual example that uses the "Architecture Browser" for examining the large scale structure of a Logical Architecture. The names vary somewhat from our presentation based example but the audience should be able to perform the mental mapping of conceptually related parts (i.e., s maps to spart and sp maps to sppart).

The left pane of this viewer shows a directory-like outline of the LA starting with an instance of a loop constraint of type "Loop2d" (i.e., the 5TH instance of this type that has been generated by DSLGen™ so far). The Loop2d5 constraint is modified by a partition set (partitionset3) which contains an edge partition APC (edge11). There are two domain variables (i.e., spart-0-edge11 and sppart-0-edge11) that are specialized to the edge11 partition. spart-0-edge11 and sppart-0-edge11 correspond to our conceptual example's specialized neighborhoods s-edge1 and sp-edge1. These two neighborhoods have been combined because they are both specialized on the same partitioning condition. That is, the generic IL forms (partestx spart-0-edge11) and (partestx sppart-0-edge11) will both refine to the same concrete form (e.g., "(== idx1 0)").

The blue square and blue call-out show what has been selected "(vbls: spart-0-edge11)" and thereby opened up in the right hand pane for examination by the domain engineer (DE). The right hand pane shows that contents of that node of the LA, which in this case is redundant with the contents of the left hand pane. If instead the DE were to select the node "(partition: edge11)", the right hand pane would reveal the non-redundant information that edge11 represents the combination of the equivalent partition sets generated earlier, edge1 and edge6.

Now, in the next slide, the DE has opened up and selected the center partition…



Logical Architecture (Internal Form)

```
(partition: edge13)
(partition: edge14)
(partition: center15)
  (vblset: (spart-0-center15 sppart-0-center15))
    (vbls: spart-0-center15)
      (transforms: col.spart-0-center15.formals)
      (transforms: row.spart-0-center15.formals)
      (transforms: qrange.spart-0-center15.formals)
      (transforms: prange.spart-0-center15.formals)
      (transforms: w.spart-0-center15.formals)
      (transforms: partest.spart-0-center15.formals)
      (transforms: partestx.spart-0-center15.formals)
    (vbls: sppart-0-center15)
      (transforms: col.sppart-0-center15.formals)
      (transforms: row.sppart-0-center15.formals)
      (transforms: qrange.sppart-0-center15.formals)
      (transforms: prange.sppart-0-center15.formals)
      (transforms: w.sppart-0-center15.formals)
      (transforms: partest.sppart-0-center15.formals)
      (transforms: partestx.sppart-0-center15.formals)
```

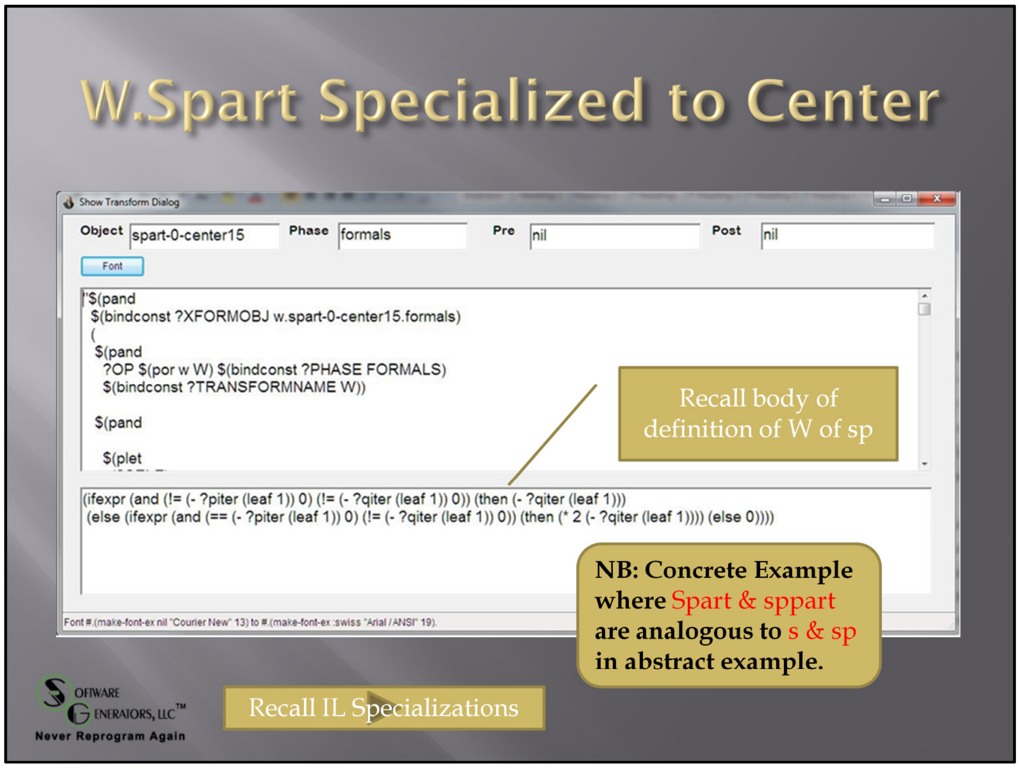
```
((|VBLS:| spart-0-center15)
 (|TRANSFORMS:|
  col.spart-0-center15.formals
  row.spart-0-center15.formals qrange.spart-0-center15.formals
  prange.spart-0-center15.formals w.spart-0-center15.formals
  partest.spart-0-center15.formals
  partestx.spart-0-center15.formals))
```

Component definitions for selected neighborhood spart-0-center15

W method-transform component definition specialized to neighborhood spart-0-center15

And this is analogous to the edge partition example of the previous slide.

All of the items in this Browser are CLOS objects and can be double clicked to open an Inspector to display their contents. For the lowest level instances (e.g., an index item like "i" or "j") this would be a Lisp inspector. However, for large scale structures that are DSLGen™ entities like transformations, the user can open a dialog specialized to transformations rather than laboriously opening a series of Lisp inspectors on all of the low level constituent parts and then trying to get the big picture perspective from those parts. For example, let us suppose that the DE wanted to examine the MT specific to the center15 partition (i.e., "w.spart-0-center15.formals"). This "triple dotted name" will be explained in the next slide.



W.Spart Specialized to Center

Show Transform Dialog

Object: spart-0-center15 Phase: formals Pre: nil Post: nil

```
'$(pand
 $(bindconst ?XFORMOBJ w.spart-0-center15.formals)
 (
  $(pand
   ?OP $(por w W) $(bindconst ?PHASE FORMALS)
   $(bindconst ?TRANSFORMNAME W))

  $(pand

   $(plet
```

```
(ifexpr (and (!= (- ?piter (leaf 1)) 0) (!= (- ?qiter (leaf 1)) 0)) (then (- ?qiter (leaf 1)))
 (else (ifexpr (and (== (- ?piter (leaf 1)) 0) (!= (- ?qiter (leaf 1)) 0)) (then (* 2 (- ?qiter (leaf 1)))) (else 0))))
```

Font #.(make-font-ex nil "Courier New" 13) to #.(make-font-ex :swiss "Arial / ANSI" 19).

Recall body of definition of W of sp

NB: Concrete Example where Spart & sppart are analogous to s & sp in abstract example.

Recall IL Specializations

SOFTWARE GENERATORS, LLC™ Never Reprogram Again

This shows a dialog that is specialized to transformation objects. A transformation object is defined by a three part name, that is,

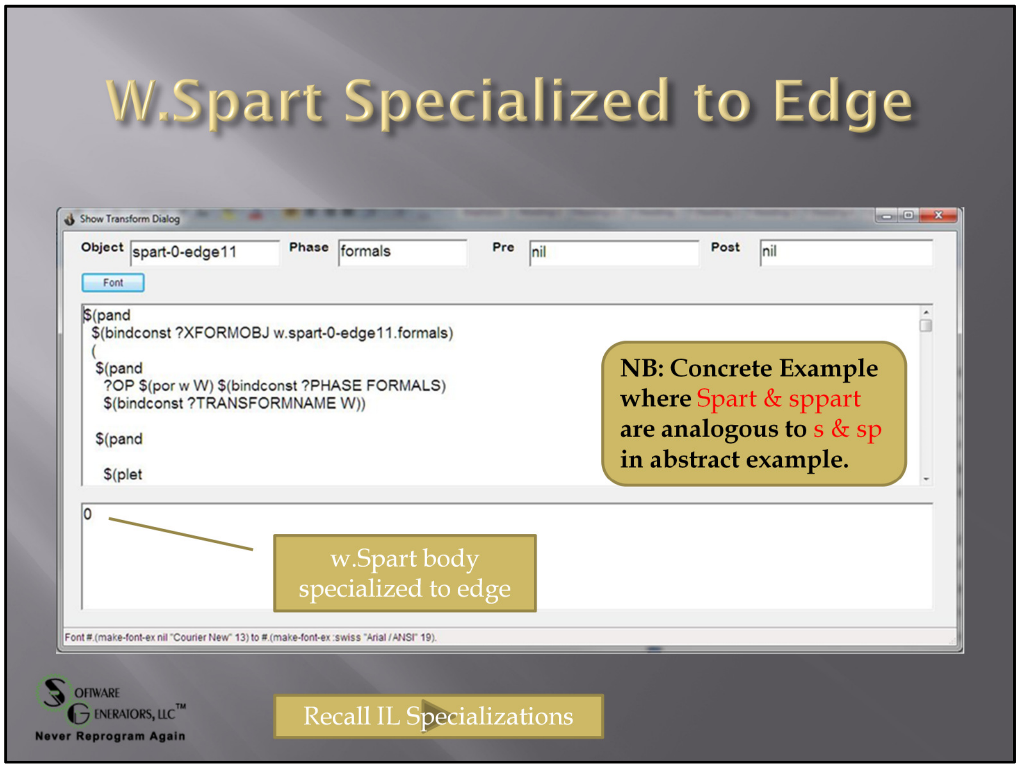
1. The method/transform name (e.g., “w”),
2. The object where its definition is stored (e.g., in this case, spart-0-center15), and
3. The generator’s phase name in which the transform is “enabled.” This means that the transform may fire ONLY during that phase (e.g., in this case, the “formals” phase, which is the generator’s phase that does definition inlining).

Transforms may also have “pre” routines that fire after a successful pattern match of the LHS but before the AST subtree is rewritten by the RHS expression.; or “post” routines that fire after the RHS re-write has occurred. These are analogous to CLOS before and after methods. Preroutines generally perform bookkeeping operations – operations that don’t lend themselves easily or efficiently to specification by transformations. MTs (in contrast to the general, process transformations that drive the generator’s overall operation) typically do not have pre or post routines and this example is typical.

The large window pane at the top is a decompiled expression of a LHS pattern that is used to match an AST subtree. These patterns have been automatically enhanced with machinery that is useful to the transformation engine but not very informative for the DE. So, we will ignore it in this example.

The **key piece of information** in this example is the RHS of the transformation, which is shown in the large window pane at the bottom. This is the internal form of the body of the center specialization of w of spart, which is the same as the center specialization of the w of sp definition that we saw in the slide illustrating the application programmer’s problem domain specific specification of the Sobel example. **(Optionally, to temporarily go to IL Specializations slide use “Recall IL Specializations” action button. The target slide is an animation, so several clicks are required to get to the Center specialization. Use the Return action button to return to this slide when done with that slide.)**

In comparison, let’s examine w.spart specialized to an edge. **Advance to next slide.**



W.Spart Specialized to Edge

Show Transform Dialog

Object spart-0-edge11 Phase formals Pre nil Post nil

Font

```
$(pand
 $(bindconst ?XFORMOBJ w.spart-0-edge11.formals)
 (
  $(pand
   ?OP $(por w W) $(bindconst ?PHASE FORMALS)
   $(bindconst ?TRANSFORMNAME W))

  $(pand

   $(plet
```

```
0
```

Font #.(make-font-ex nil "Courier New" 13) to #.(make-font-ex :swiss "Arial / ANSI" 19).

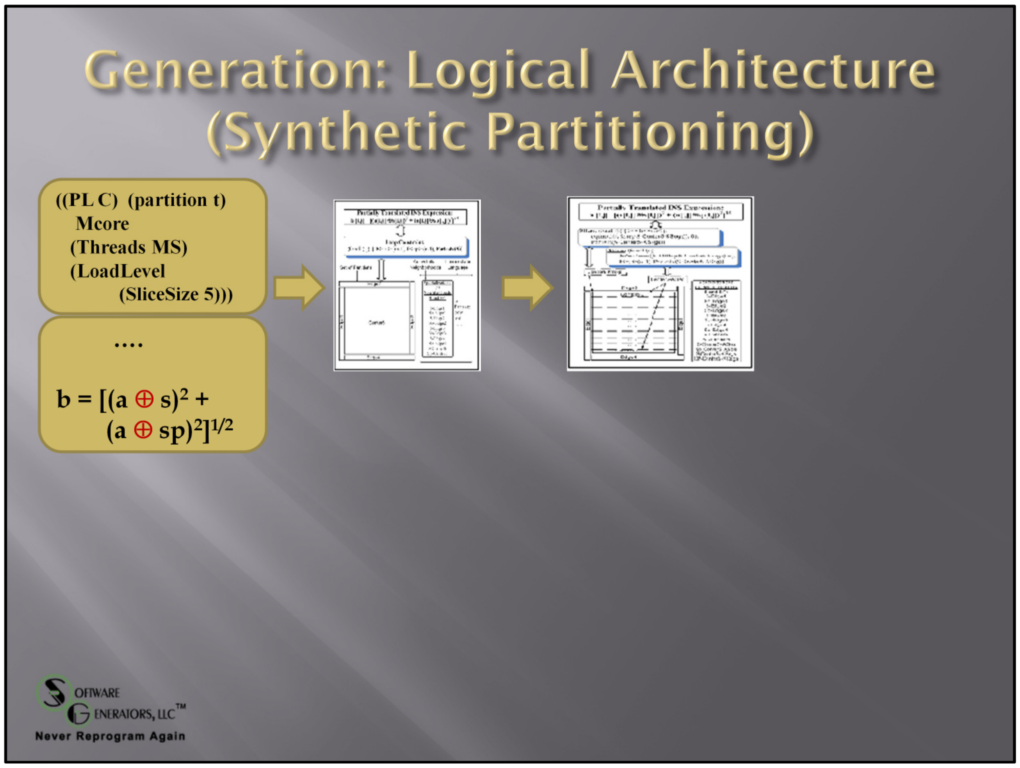
NB: Concrete Example where Spart & sppart are analogous to s & sp in abstract example.

w.Spart body specialized to edge

Recall IL Specializations

By contrast to the center specialization of w, an edge specialization produces a RHS of 0. When the problem domain expressions specific to this edge are in lined with their definitions and then partially evaluated, this value will cause the arithmetic expression containing a instance of w.spart for this edge to partially evaluate to zero. That zero result will be one of the zeros on the RHS of the assignment statement within the edge processing loops. In other words, this definition is the key to the simplification of the pixel values for edge pixels in all three code examples we examined earlier.

(As with the previous slide, if the speaker wants to forcefully make the connection to the MT specialization step shown earlier, he can temporarily go to the IL Specializations slide. Use "Recall IL Specializations" action button. The target slide is an animation, so several clicks are required to get to the Edge specialization. Use the Return action button to return to this slide when done with the IL Specializations slide.)



Generation: Logical Architecture (Synthetic Partitioning)

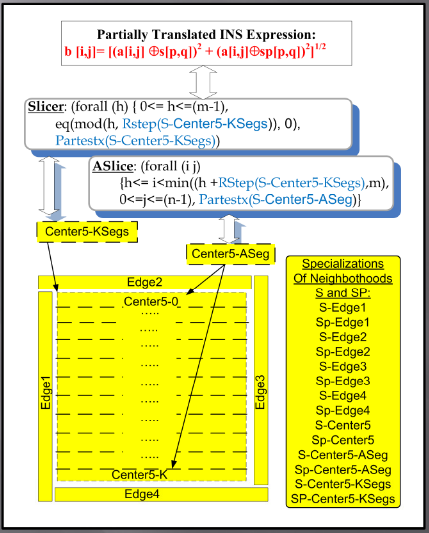
```
((PL C)  (partition t)
   Mcore
  (Threads MS)
  (LoadLevel
         (SliceSize 5)))
```

....

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

The next step in generation is introducing elective design features based on the execution platform specification. An example of such a design feature would be the division of heavyweight computation partitions into sub-partitions that can be executed in parallel (e.g., slicing the image center pixels into center slices). Such design features will be added by 1) introducing so-called synthetic partitions that imply the design feature and 2) revising and reorganizing of other APCs appropriately. A conceptual view of an LA after synthetic partitioning with its accompanying revisions is shown in the next slide.

Generation: Logical Architecture (Synthetic Partitioning)



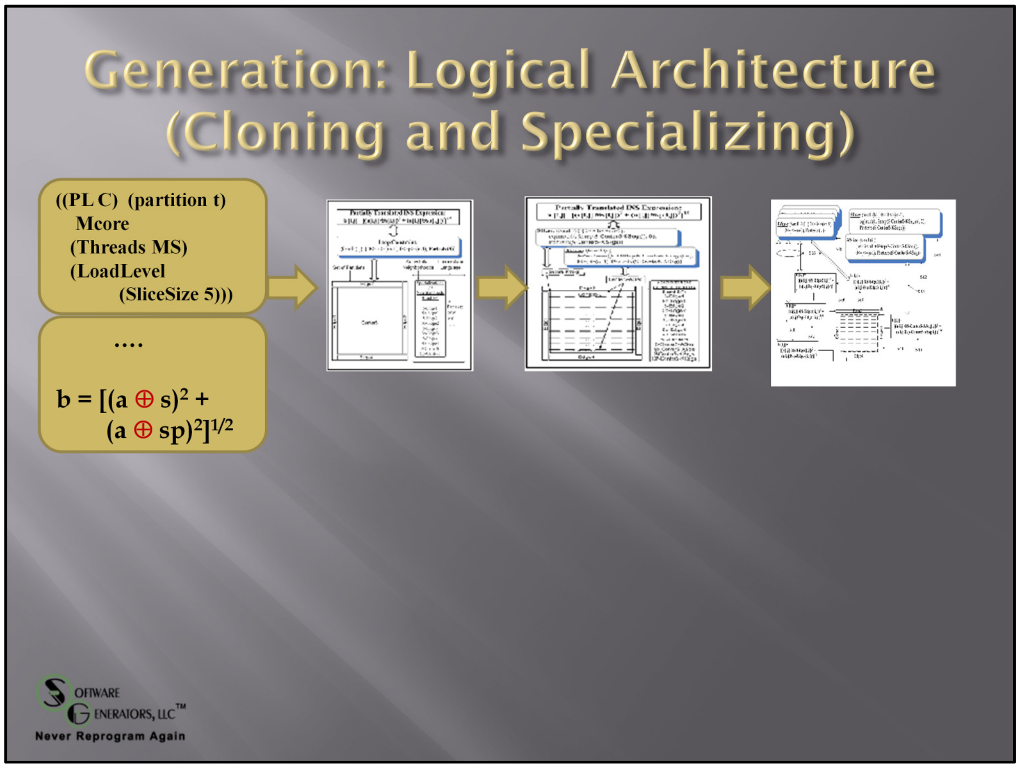
Partially Translated INS Expression:

$b\ [i,j] = [(a[i,j] \oplus s[p,q])^2 + (a[i,j] \oplus sp[p,q])^2]^{1/2}$

```
Slicer: (forall (h) { 0<= h<=(m-1),
      eq(mod(h, Rstep(S-Center5-KSegs)), 0),
      Partestx(S-Center5-KSegs))

ASlice: (forall (i j)
      {h<= i<min((h +RStep(S-Center5-KSegs),m),
      0<=j<=(n-1), Partestx(S-Center5-ASeg)}
```

The center partition has been revised into two specializations: 1) Center5-Ksegs representing the whole center partition after it has been sliced into groups of image rows, and 2) Center5-ASeg representing an instance of a slice of the center. Simultaneously, the loop APC has been revised into two loops: one (over the newly created variable "h") that calculates the index of the first row of image pixels for each slice and a second loop APC that processes through the rows of one center slice. The new center neighborhoods have been specialized to these new partitions as have their MT component definitions, if there are any component dependencies on the new partitions. In this case the MT definitions of w of the neighborhoods are unchanged from s-Center5 and sp-Center5 specializations of the neighborhoods, so MT definitions for the w method will be inherited from s-Center5 and sp-Center5. However, MT definitions for Partestx (which refines to some specific partitioning condition) and Rstep (which refines to a loop increment value), are dependent on the new neighborhood specializations, so new MT definitions are generated for those neighborhoods.

That completes the LA of the partially translated INS expression so DSLGen™ now has enough information to clone specialized versions of the partially translated INS expression.



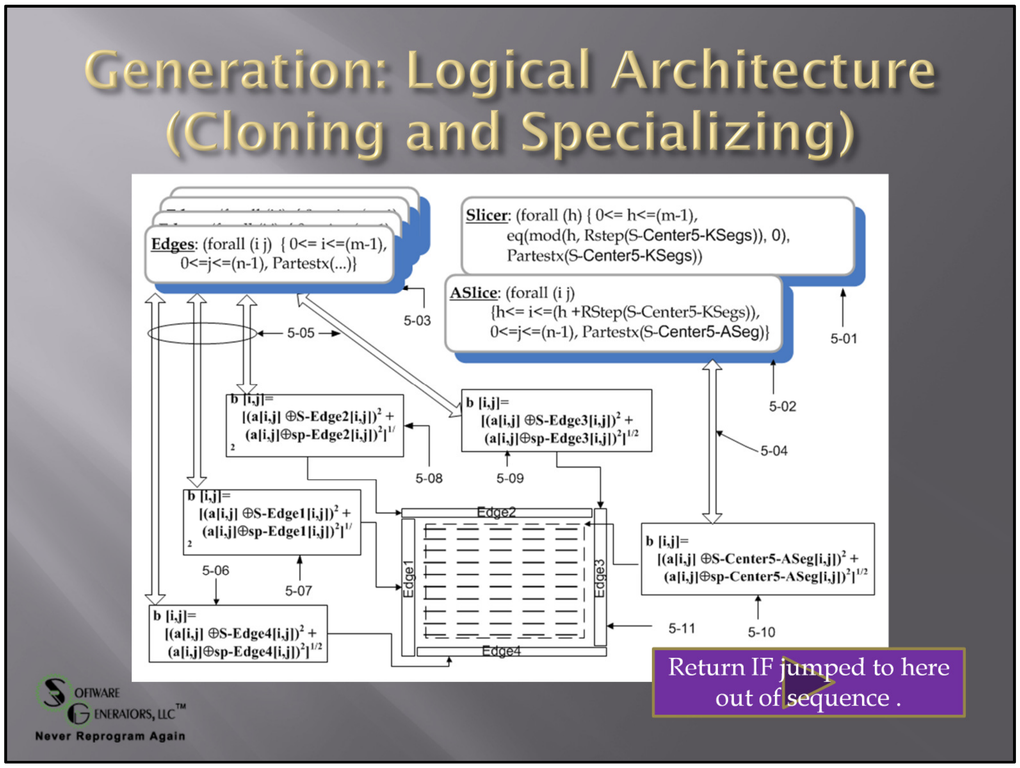
Generation: Logical Architecture (Cloning and Specializing)

```
((PL C)  (partition t)
   Mcore
  (Threads MS)
  (LoadLevel
         (SliceSize 5)))
```

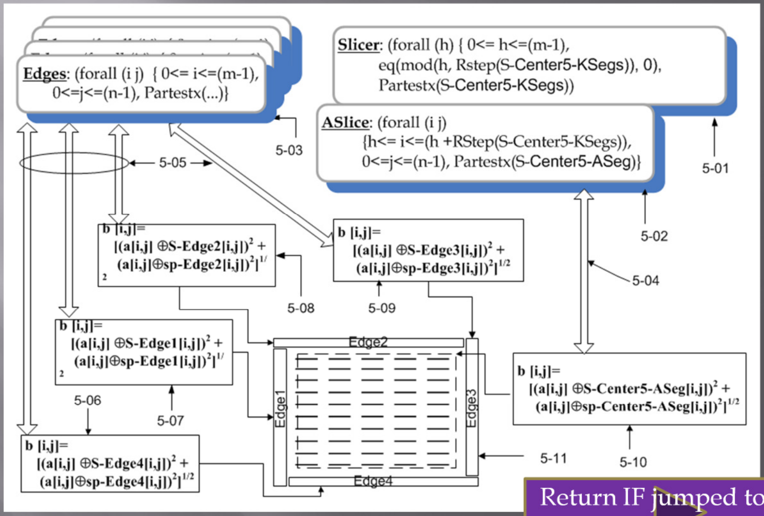
....

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

Cloning and specialization of the INS expression is achieved by applying the LA to the INS expression thereby producing the following clones.



Generation: Logical Architecture (Cloning and Specializing)



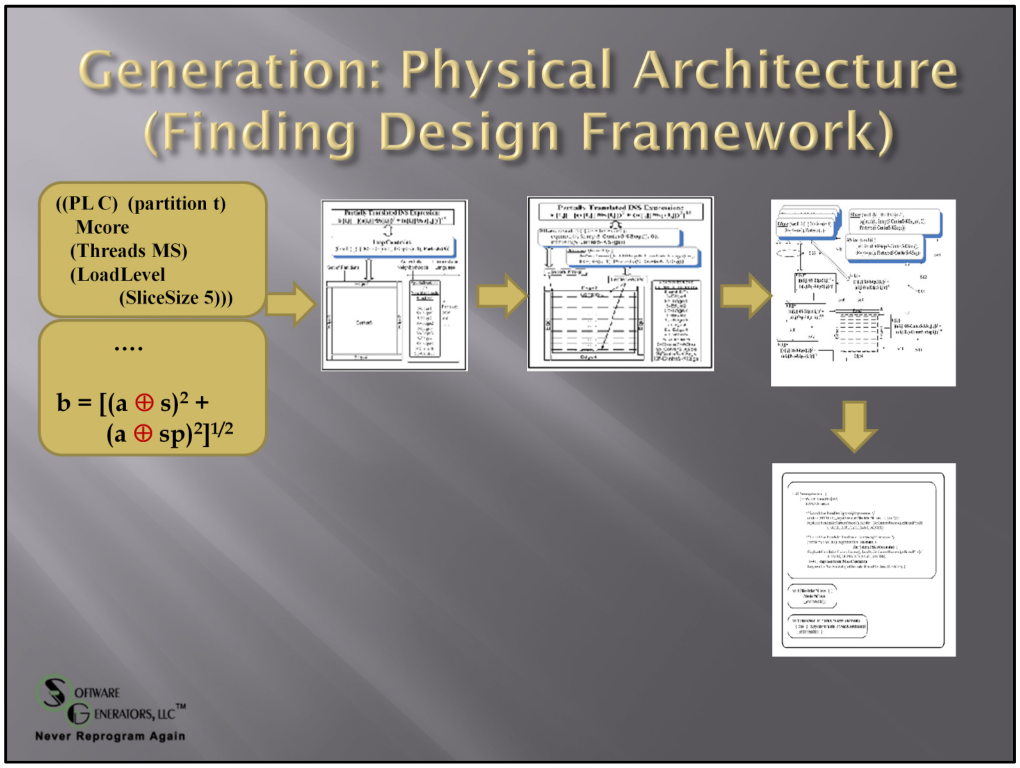
Edges: (forall (i j) { 0<= i<=(m-1), 0<=j<=(n-1), Partestx(...)}
Slicer: (forall (h) { 0<= h<=(m-1), eq(mod(h, Rstep(S-Center5-KSegs)), 0), Partestx(S-Center5-KSegs))
ASlice: (forall (i j) {h<= i<=(h +RStep(S-Center5-KSegs)), 0<=j<=(n-1), Partestx(S-Center5-ASeg)}
5-01
5-02
5-03
5-04
5-05
b [i,j]= [(a[i,j] ⊕S-Edge2[i,j])² + (a[i,j]⊕sp-Edge2[i,j])²]^1/2
b [i,j]= [(a[i,j] ⊕S-Edge3[i,j])² + (a[i,j]⊕sp-Edge3[i,j])²]^1/2
5-08
5-09
b [i,j]= [(a[i,j] ⊕S-Edge1[i,j])² + (a[i,j]⊕sp-Edge1[i,j])²]^1/2
Edge2
Edge1
Edge3
Edge4
b [i,j]= [(a[i,j] ⊕S-Center5-ASeg[i,j])² + (a[i,j]⊕sp-Center5-ASeg[i,j])²]^1/2
5-06
5-07
b [i,j]= [(a[i,j] ⊕S-Edge4[i,j])² + (a[i,j]⊕sp-Edge4[i,j])²]^1/2
5-11
5-10

Return IF jumped to here out of sequence .

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

This diagram represents a conceptual view of the clones (i.e., 5-06 through 5-10) of the partially translated problem domain spec, their relationship to the various partitions and their association to the specialized forms of the loops. But where do these specialized loops and clones go in the overall design of the target program? What is the context that receives these elements?

What DSLGen™ does NOT YET HAVE is an overall design framework that will provide the context for and the interstitial structure to receive and correctly integrate these various clones and loops. However, based on the execution platform spec descriptors and the nature of the computation, the synthetic design process has very purposely built a logical architecture that is aimed at a specific class of design frameworks (including the one we will use, which is named the Thread-Based Slicer/Slicee design framework). This design framework will provide that context and interstitial structure.



Generation: Physical Architecture (Finding Design Framework)

((PL C) (partition t)
Mcore
(Threads MS)
(LoadLevel
(SliceSize 5)))

....

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

Partially Translated INS Expression
Partially Translated INS Expression
Oftware Generators, LLC™
Never Reprogram Again

A design framework is a generalization and formalization of the "gang of four's" notion of a design pattern. (See “Design Patterns”, Gamma, et al, Addison Wesley.)

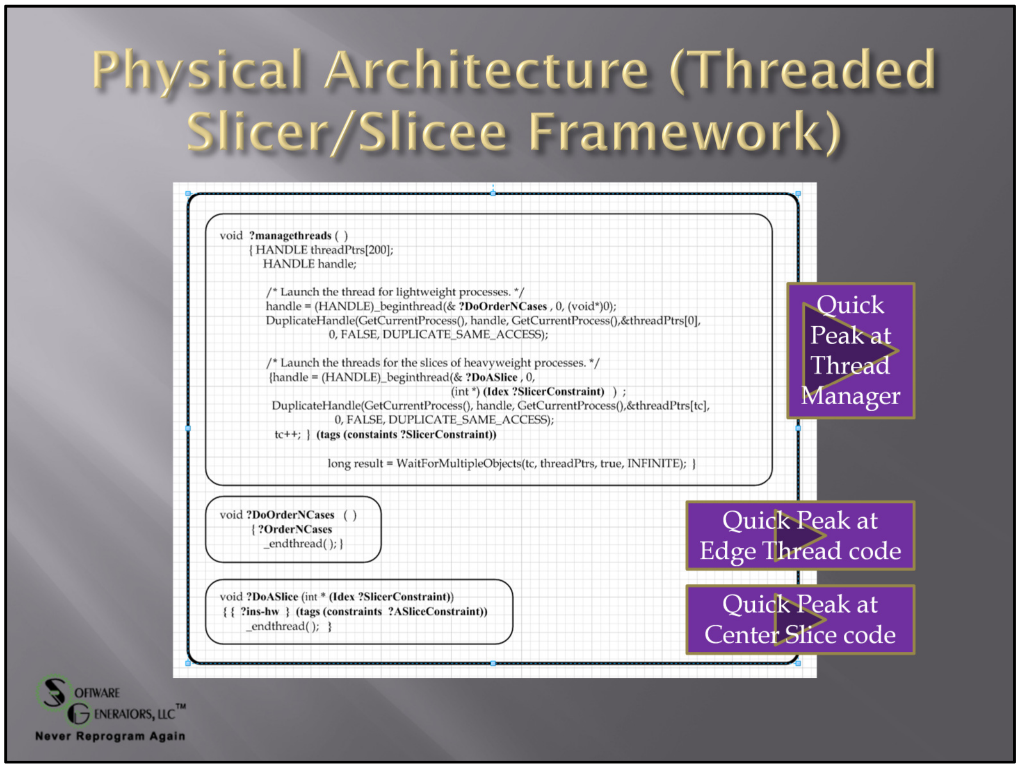
Design frameworks provide basic facilities:

1. **Global architectural structures** containing “holes” to receive the cloned and specialized elements of the computation specification (e.g., in this case, these are interrelated thread routine skeletons and global thread synchronization patterns) and

2. **Low level coding clichés** (e.g., portions of those frameworks might include thread clichés to initiate threads and synchronization steps to synchronize sets of threads), and

3. Expressions that specify and abstractly reference **design and code entities from the logical architecture** (e.g., **?vbls** pre-computed by the design framework method such as generated routine names; APCs specific to the framework such as the constraints identifying the Slicer and Slicee constraints; clone-specific implementation neutral specification (INS) such as ?ins-hw (i.e., heavyweight is center slice) or ?ins-lw (i.e., lightweight will be the set of edges pixel computations). These ?vbls play the role of parameters for the design framework.

Now, let’s look at an example of a design framework for the thread-based slicer/slicee design.



Physical Architecture (Threaded Slicer/Slicee Framework)

```
void ?managethreads ( )
      { HANDLE threadPtrs[200];
        HANDLE handle;

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

          /* Launch the threads for the slices of heavyweight processes. */
          {handle = (HANDLE)_beginthread(& ?DoASlice , 0,
                                    (int *) (Idex ?SlicerConstraint)  ) ;
           DuplicateHandle(GetCurrentProcess(), handle, GetCurrentProcess(),&threadPtrs[tc],
                    0, FALSE, DUPLICATE_SAME_ACCESS);
           tc++; } (tags (constaints ?SlicerConstraint))

                    long result = WaitForMultipleObjects(tc, threadPtrs, true, INFINITE); }

void ?DoOrderNCases  ( )
      { ?OrderNCases
         _endthread( ); }

void ?DoASlice (int * (Idex ?SlicerConstraint))
 { { ?ins-hw } (tags (constraints ?ASliceConstraint))
     _endthread( ); }
```

Quick Peak at Thread Manager
Quick Peak at Edge Thread code
Quick Peak at Center Slice code
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

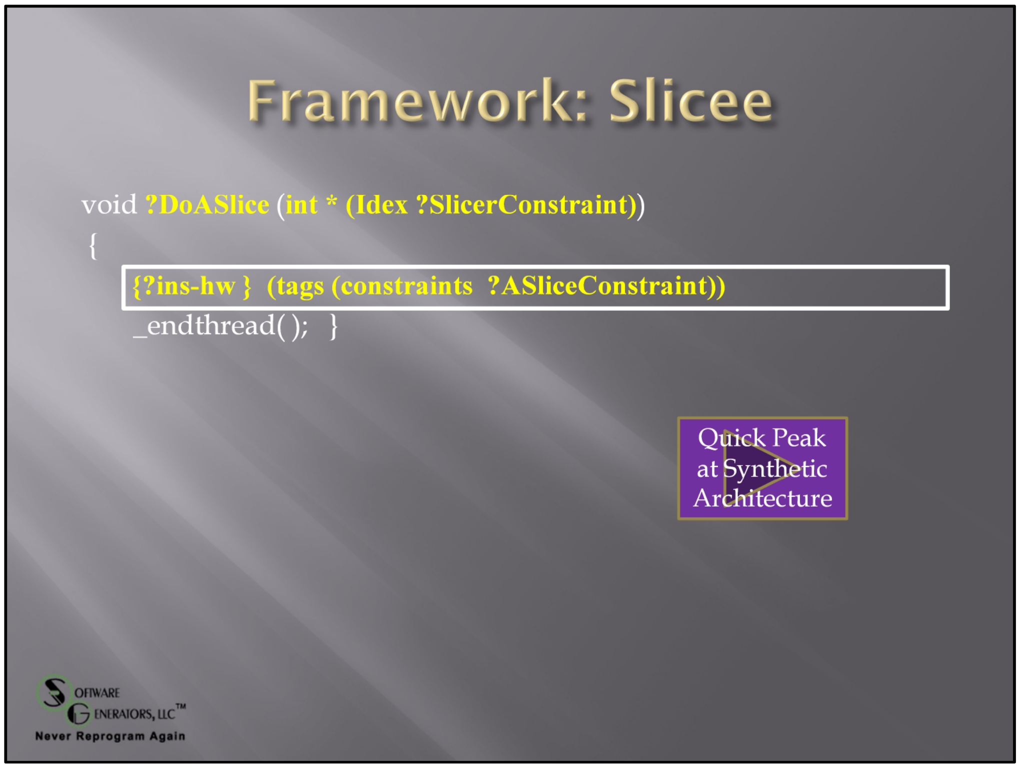
As mentioned earlier, the synthetic design process has very purposely built a logical architecture that is aimed at a set of designed frameworks including this specific framework (named the Thread-Based Slicer/Slicee design framework) that will provide the architectural and interstitial structure for the resultant generated code.

This figure clearly shows the architectural structure of the three routines that comprise the thread-based Slicer/Slicee Design: 1. The thread manager routine (with a generated name of "SobelThreads8"), 2. The lightweight thread that will batch process all of the edge cases (with a generated name of "SobelThreads9"), and 3. The Center Slice code (with a generated name of "SobelCenterSlice10").

(Speaker: The action buttons are provided to jump to the actual code used to describe design features earlier in this presentation. There are return buttons on the actual Slicer/Slicee code slides so it is easy for the audience to see the relationship between the abstract references to elements of the LA, e.g., ?ins-hw and ?managethreads, and the actual code. These design/code comparisons provide an opportunity for the audience to get a clear intuition about the source of code features. Some arise inexorably from the computational essence and some arise from the elective design features specified by the application programmer. Comparison can make this point in a highly intuitive way and can be used with the conceptual description below.)

Notice that the design framework for the edge cases and the center cases provides precious little real code level detail. They largely define the architectural structure (i.e., holes) into which the real code details provided by the cloned and specialized versions of the INS are to be put. If you think about it, that makes perfect sense because most of the code of these routines are in fact determined by the essence of the computation (provided by the domain specific INS specification). The routine structure, pattern of thread synchronization, and the parameter structure of the center slice routine arise from elective design features that are imposed on the computation because of the requirements contained within the execution platform description.

In the next few slides, we will follow the steps of the inlining process that will integrate the center thread routine ("SobelCenterSlice10", which the method associated with the slicer/slicee design framework has bound to the variable ?DoASlice) .



Framework: Slicee

```
void ?DoASlice (int * (Idex ?SlicerConstraint))
 {
    {?ins-hw }  (tags (constraints  ?ASliceConstraint))
    _endthread( );  }
```

Quick Peak at Synthetic Architecture
Software Generators, LLC™
Never Reprogram Again

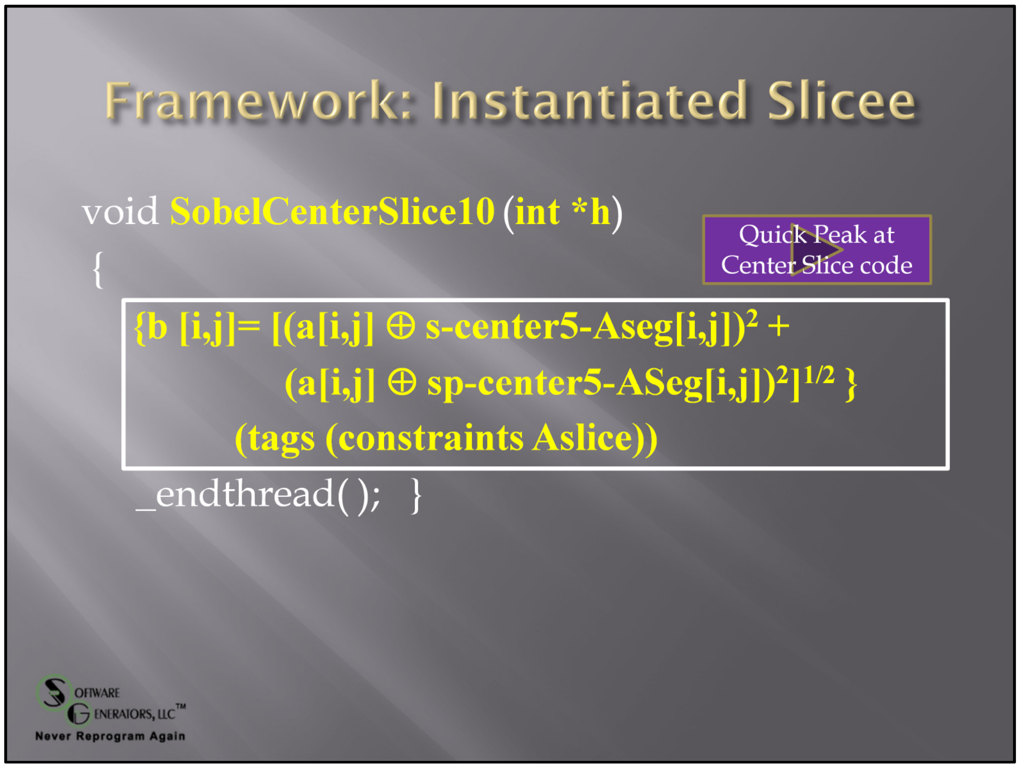
Each design framework has a method routine that does a lot of the bookkeeping work. The simplest bookkeeping job is the generation of unique names for the routines in the Slicer/Slicee framework and the binding of them to design framework specific variables (i.e., ?vbls variables) that the framework methods use to stand in for those names. Recall that the generated name for this routine in our example is “SobelCenterSlice10”. It will be bound to ?DoASlice.

The domain concepts that are common between the framework’s skeletal forms and the DF method are represented in the DF by ?vbls or meta-operations (e.g., Idex) on ?vbls. In fact, all of the text represented here in yellow represents a “common language or protocol” for referencing entities in the LA as well as some of entities created by the design framework (e.g., the routine names). Some of these ?vbls represent the expected instances of specialized clones and loops. And the domain knowledge provides the generator sufficient clues to pin down each expected entity shown in the previous slide titled Cloning and Specialization. **(Speaker: Use the “quick peak…” action button to jump to the slide showing the specialized clones and point out the center slice clone (the domain specific expression specialized to s-Center5-Aseg and sp-Center5-Aseg) and its associated loop constraint (Aslice). Note that the other loop (Slicer) will be referenced from the protocol expressions in the thread manager routine (SobelThreads8). Then return to this slide with the return action button.)**

To be more specific, the DF sets up the meta-information by binding of ?ins-hw to the clone that is the ?ins specialized to the heavyweight computation (i.e., the center partition computation) . If we look back at the Cloning and Specialization slide we will see that binding value will be “**b [i,j]= [(a[i,j] ⊕ s-center5-Aseg[i,j])^2 + (a[i,j] ⊕ sp-center5-ASeg[i,j])^2]$^{1/2}$** “ Similarly, ?slicerconstraint will get bound to the “Aslice” loop APC, which computes the starting index of each center slice, and the search expression (idex ?slicerconstraint) will refine to the name of the target program index variable of that loop, i.e., “h”. Finally (for the Slicee), ?ASliceConstraint will get bound to the loop APC that processes a center slice (i.e., Aslice) .

Other ?vbls will get set up to handle the thread routine that handles to the edges and the thread manager routine but we will address those bindings when we show the evolution of those routines.

So, with the meta-information bindings we have identified so far, let’s follow the refinement of the ?DoASlice routine to code. **…Next slide …**



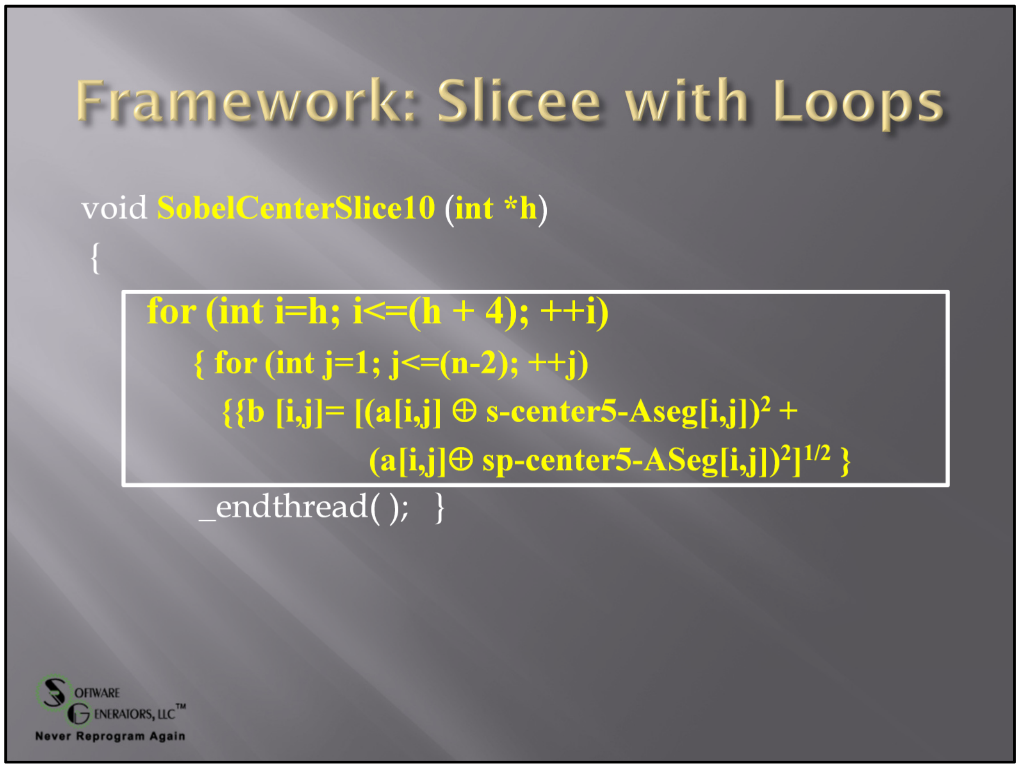
Framework: Instantiated Slicee

```
void SobelCenterSlice10 (int *h)
{
    {b [i,j]= [(a[i,j] ⊕ s-center5-Aseg[i,j])^2 +
                 (a[i,j] ⊕ sp-center5-ASeg[i,j])^2]^1/2 }
            (tags (constraints Aslice))
    _endthread( );  }
```

Quick Peak at Center Slice code
OFTWARE ENERATORS, LLC™
Never Reprogram Again

All that has happened here is the inlining of routine name, it's parameter and the INS clone specific to the center slice. Now, *more or less* all that is left to do is to generate the loop structure from the Aslice constraint, recursively inline the operator definitions and do simplification via partial evaluation. (Actually, there is still a need for some tidying up of definitions for this scope, generating some comments to leave a forensic trail for the domain engineer in case of bugs as well as provide a bit of insight for the application programmer, and so forth. Some of the remaining steps are tying up loose ends and dealing with cosmetics like code pretty printing.)

Now that "*more or less*" comment is really "*more*" than "*less*" but getting into it is getting too far into the weeds. Suffice it to say, that the convolution operations on pixels has associated logical architectures (i.e., loop constraints) that will cause the generation of the neighborhood loops. **(Speaker: Take a quick peek back at the Center Slice code and point out the loops over p and q as the result of the LA associated with convolution over pixels operations. Then use the return action button to return to this slide.)**



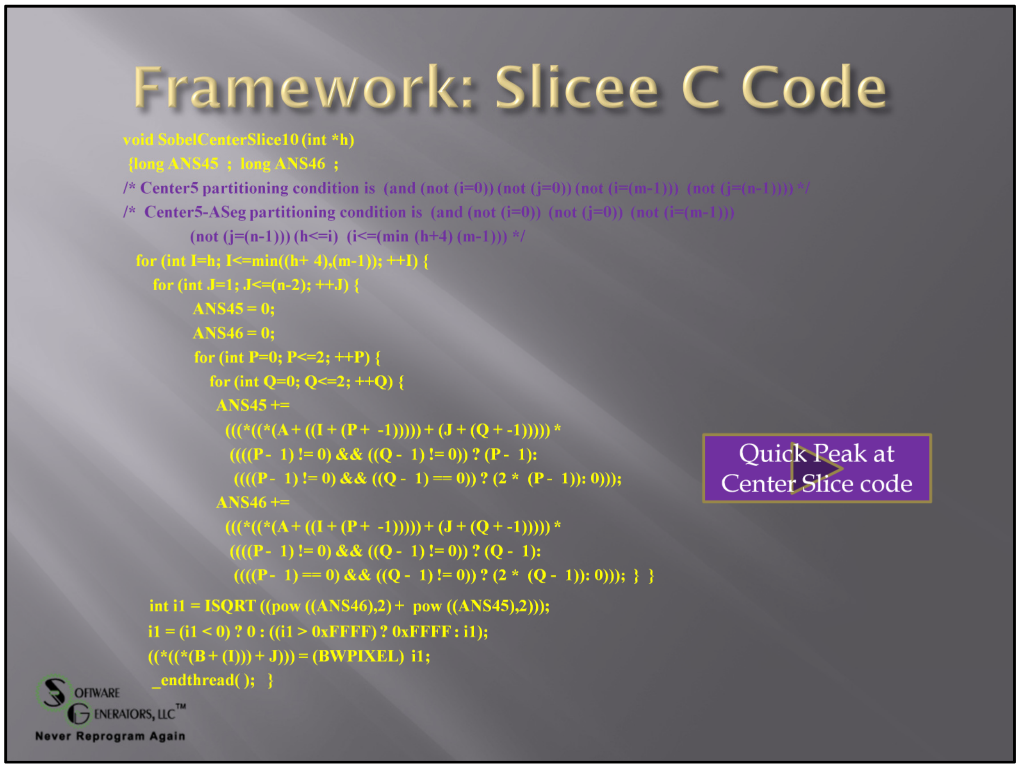
Framework: Slicee with Loops

```
void SobelCenterSlice10 (int *h)
{
    for (int i=h; i<=(h + 4); ++i)
        { for (int j=1; j<=(n-2); ++j)
          {{b [i,j]= [(a[i,j] ⊕ s-center5-Aseg[i,j])^2 +
                      (a[i,j]⊕ sp-center5-ASeg[i,j])^2]^1/2 }
      _endthread( );  }
```

OFTWARE
ENERATORS, LLC™
Never Reprogram Again

After generating the loop code from the Aslice APC, we will evolve to code that looks like the code in this slide.

After further inlining, partial evaluation and some tidying up, the final code will look something like the next slide. **… Next Slide …**



Framework: Slicee C Code

```
void SobelCenterSlice10 (int *h)
 {long ANS45  ;  long ANS46  ;
/* Center5 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 +=
                  (((*((*(A + ((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 +=
                  (((*((*(A + ((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);
     ((*((*(B + (I))) + J))) = (BWPIXEL)  i1;
      _endthread( );   }
```

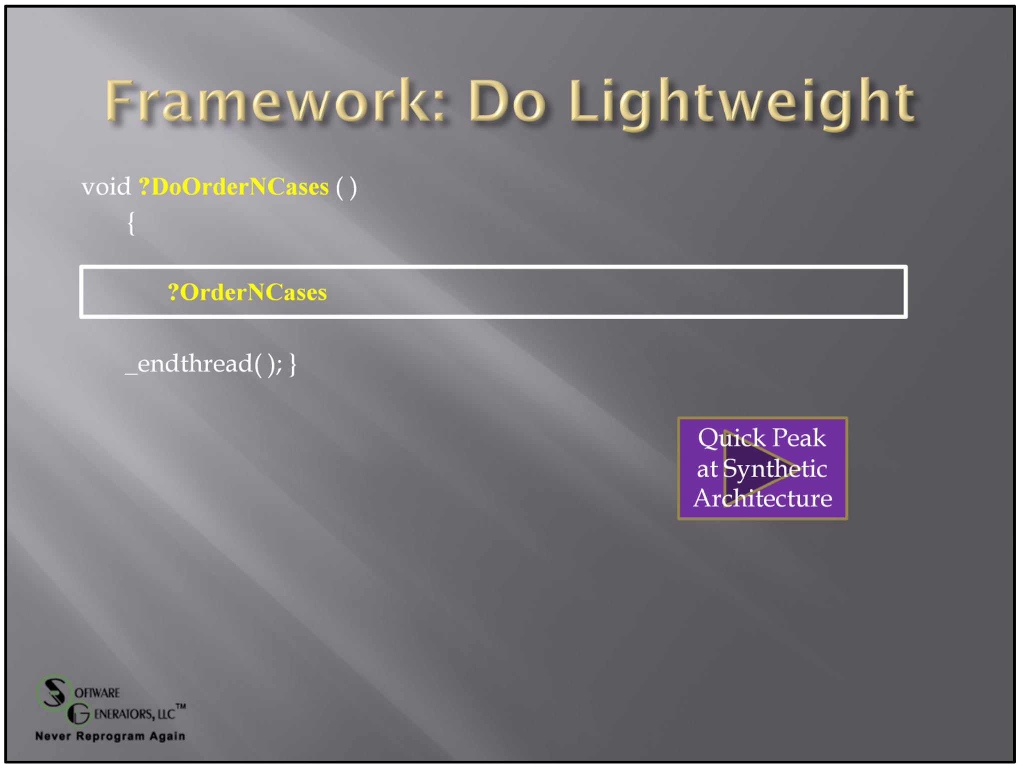
Quick Peak at
Center Slice code

SOFTWARE
GENERATORS, LLC™
Never Reprogram Again

The final code will look something like this (after IL inlining and partial evaluation).

(If the audience wants/needs to compare this to the code we looked at earlier that had all of the DESIGN FEATURES identified by CALLOUTS, use the Quick Peak action button and when done return to this slide via the Return Action button on the Quick Peak target page.)

Now, let’s examine how the lightweight clones get integrated into the design framework.

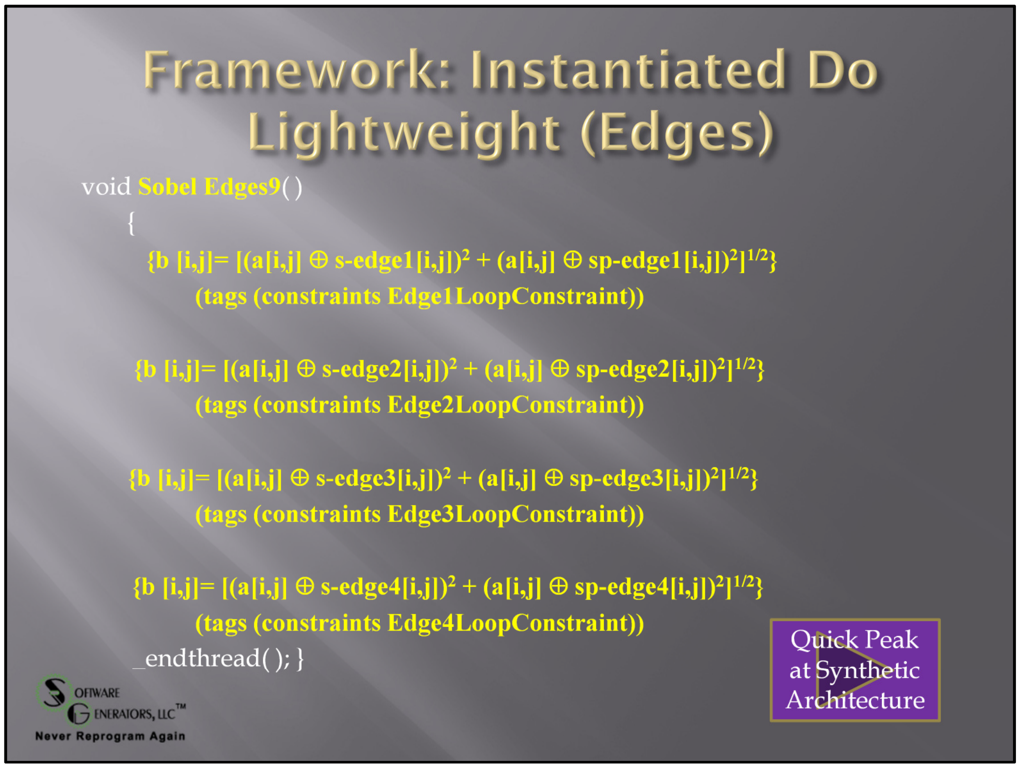


Framework: Do Lightweight

```
void ?DoOrderNCases ( )
      {
            ?OrderNCases

      _endthread( ); }
```

Quick Peak at Synthetic Architecture
OFTWARE
ENERATORS, LLC™
Never Reprogram Again

This is the framework skeleton for lightweight processes. Recall that the generator made the decision to batch the edge computations into a single thread based on the domain knowledge about the "edge" class that edge's are lightweight computations (i.e., likely to be order(n) computations rather than something like order(n^2) computations). In this case, ?OrderNCases will be bound to the set of edge clones (diagram items 5-06 through 5-09) and their associated loop constraints (diagram item 5-03) from the Cloning and Specializing slide. **(Speaker: Quick peak back to refresh the listener's mind, if necessary.)**



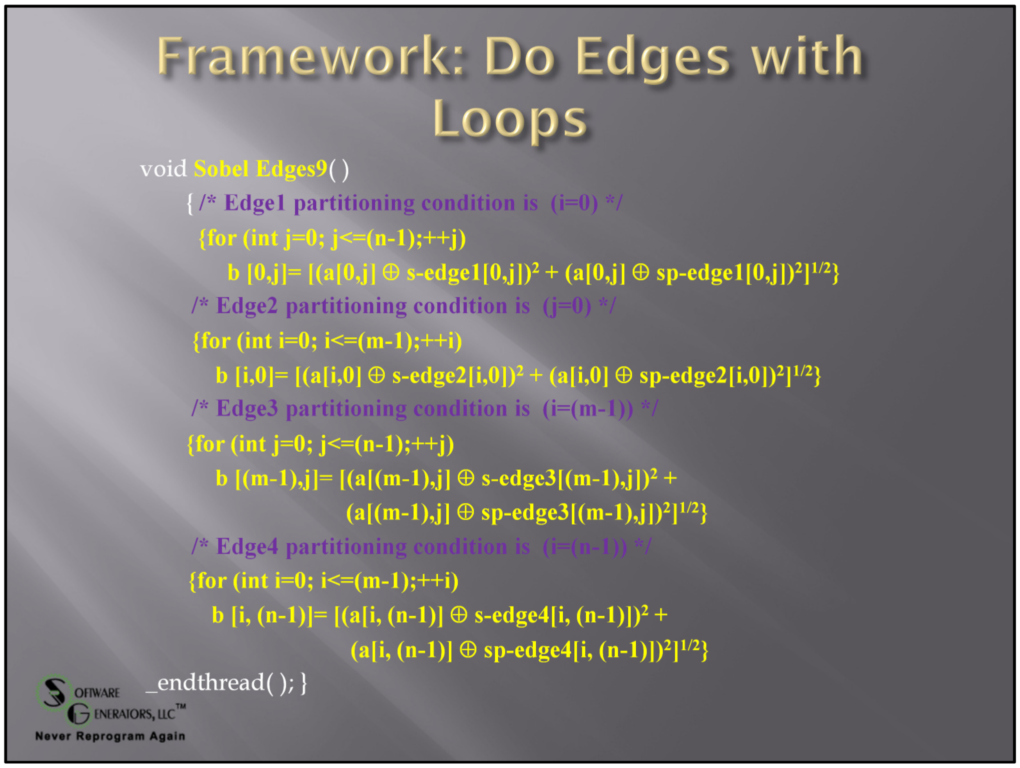
Framework: Instantiated Do Lightweight (Edges)

```
void Sobel Edges9( )
    {
      {b [i,j]= [(a[i,j] ⊕ s-edge1[i,j])$^2$ + (a[i,j] ⊕ sp-edge1[i,j])$^2$]$^{1/2}$}
            (tags (constraints Edge1LoopConstraint))

      {b [i,j]= [(a[i,j] ⊕ s-edge2[i,j])$^2$ + (a[i,j] ⊕ sp-edge2[i,j])$^2$]$^{1/2}$}
            (tags (constraints Edge2LoopConstraint))

      {b [i,j]= [(a[i,j] ⊕ s-edge3[i,j])$^2$ + (a[i,j] ⊕ sp-edge3[i,j])$^2$]$^{1/2}$}
            (tags (constraints Edge3LoopConstraint))

      {b [i,j]= [(a[i,j] ⊕ s-edge4[i,j])$^2$ + (a[i,j] ⊕ sp-edge4[i,j])$^2$]$^{1/2}$}
            (tags (constraints Edge4LoopConstraint))
    _endthread( ); }
```

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Quick Peak at Synthetic Architecture

Substituting the edge clones and their loop constraints we get the form shown here. **… next slide …**

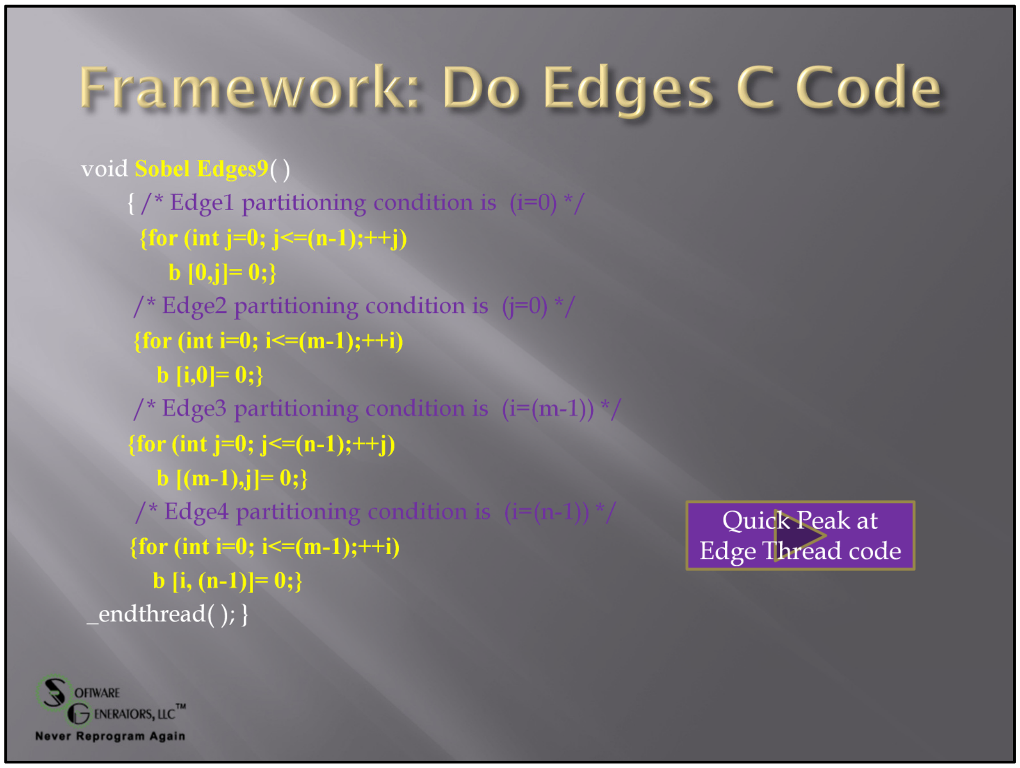


Framework: Do Edges with Loops

```
void Sobel Edges9( )
      { /* Edge1 partitioning condition is  (i=0) */
        {for (int j=0; j<=(n-1);++j)
            b [0,j]= [(a[0,j] ⊕ s-edge1[0,j])² + (a[0,j] ⊕ sp-edge1[0,j])²]^1/2}
       /* Edge2 partitioning condition is  (j=0) */
       {for (int i=0; i<=(m-1);++i)
          b [i,0]= [(a[i,0] ⊕ s-edge2[i,0])² + (a[i,0] ⊕ sp-edge2[i,0])²]^1/2}
       /* Edge3 partitioning condition is  (i=(m-1)) */
      {for (int j=0; j<=(n-1);++j)
          b [(m-1),j]= [(a[(m-1),j] ⊕ s-edge3[(m-1),j])² +
                            (a[(m-1),j] ⊕ sp-edge3[(m-1),j])²]^1/2}
       /* Edge4 partitioning condition is  (i=(n-1)) */
      {for (int i=0; i<=(m-1);++i)
          b [i, (n-1)]= [(a[i, (n-1)] ⊕ s-edge4[i, (n-1)])² +
                            (a[i, (n-1)] ⊕ sp-edge4[i, (n-1)])²]^1/2}
   _endthread( ); }
```

SOFTWARE GENERATORS, LLC™ Never Reprogram Again

Analogous to the center loop inlining, we inline the edge loops and refine the expressions to the point of convolution operations at the pixel level, i.e., convolutions over the neighborhood of pixel a[i,j] where i and j are specialized appropriately for the four edge cases. Further inlining for the neighborhood loops and simplification leads to the next slide.



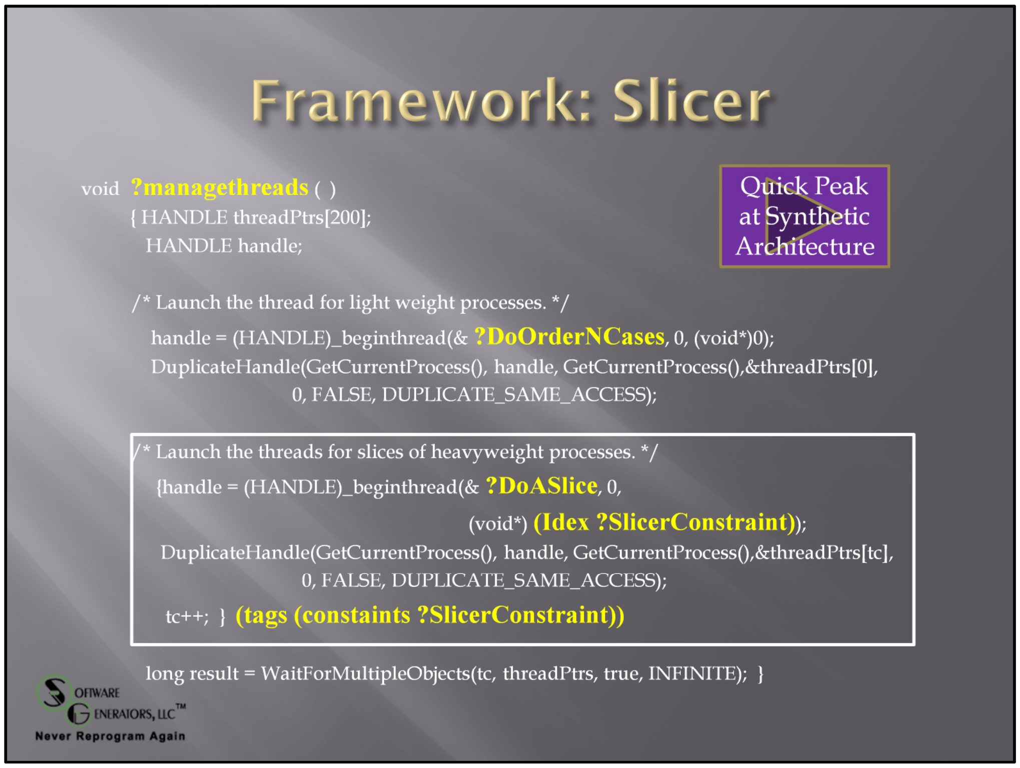
Framework: Do Edges C Code

```
void Sobel Edges9( )
      { /* 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  (i=(n-1)) */
      {for (int i=0; i<=(m-1);++i)
          b [i, (n-1)]= 0;}
 _endthread( ); }
```

Quick Peak at
Edge Thread code
OFTWARE
ENERATORS, LLC™
Never Reprogram Again

Note that we have glossed over how the RHS of the assignment statements get simplified to zero. The short story is that lining the definitions for convolution and recursively for its component operations produces arithmetic expressions multiplied times zero (i.e., the value of W for the various edge specific neighborhoods) which partially evaluate to zero. Note: The tool demo presentation associated with this presentation provides a sort of "animation" of this simplification for one specific case if you want to see the process at its most basic level. The take home point is that by specializing the MT definitions of operators to specific partitions, the code produced for those partitions will be the correct code for the context of the specific partition.

..next slide..



Framework: Slicer

```
void ?managethreads ( )
     { HANDLE threadPtrs[200];
       HANDLE handle;

     /* Launch the thread for light weight processes. */
       handle = (HANDLE)_beginthread(& ?DoOrderNCases, 0, (void*)0);
       DuplicateHandle(GetCurrentProcess(), handle, GetCurrentProcess(),&threadPtrs[0],
                      0, FALSE, DUPLICATE_SAME_ACCESS);

     /* Launch the threads for slices of heavyweight processes. */
       {handle = (HANDLE)_beginthread(& ?DoASlice, 0,
                                         (void*) (Idex ?SlicerConstraint));
        DuplicateHandle(GetCurrentProcess(), handle, GetCurrentProcess(),&threadPtrs[tc],
                       0, FALSE, DUPLICATE_SAME_ACCESS);
        tc++; } (tags (constaints ?SlicerConstraint))

       long result = WaitForMultipleObjects(tc, threadPtrs, true, INFINITE); }
```

Quick Peak at Synthetic Architecture

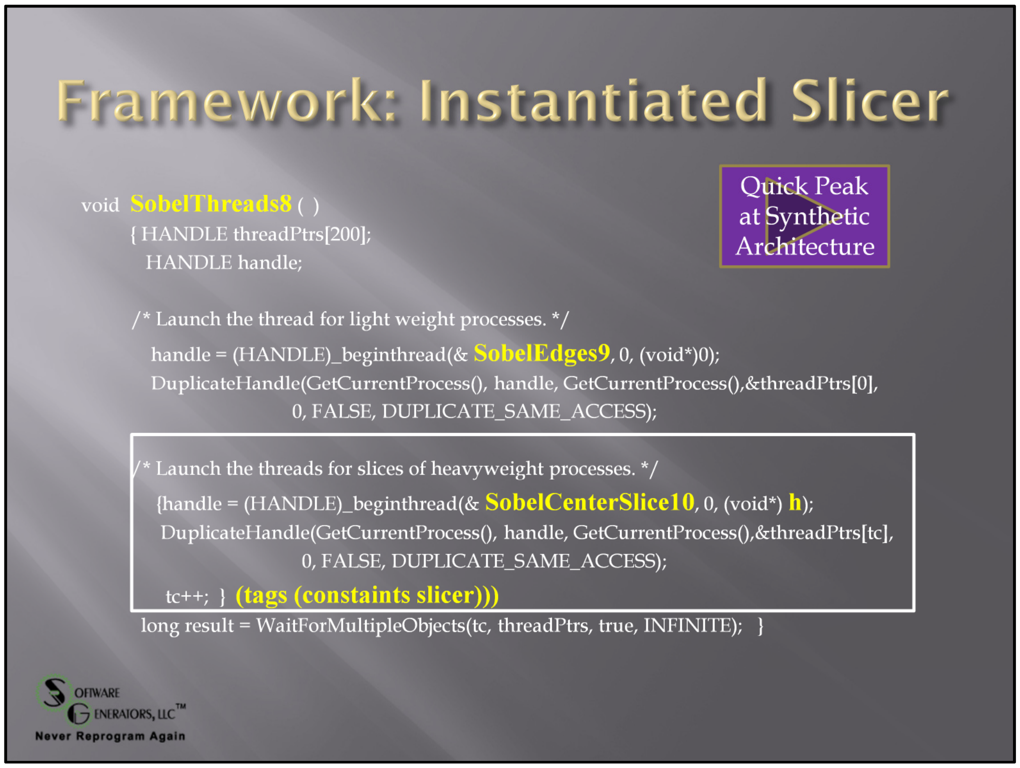
Software Generators, LLC™
Never Reprogram Again

Finally, let's examine the refinement of the thread manager routine, which will define the control flow between the thread manager and the two other thread routines. Recall that **?DoOrderNCases** will be bound to the name of the routine (i.e., "SobelEdges9") that is generated to compute the lightweight (i.e., edge) cases and **?DoASlice** will be bound to the name of the routine (i.e., "SobelCenterSlice10") that is generated to compute the heavyweight (i.e., center slice) cases. (Idex **?SlicerConstraint**) will refine to the index name (i.e., "h") generated for the **?SlicerConstraint** loop constraint (i.e., Aslice).

Recall that the expressions in yellow font represent the protocol for referring to entities that are expected to be in the Logical Architecture but do not yet exist when the protocol expressions are written.

Additionally, the Slicer design provides the patterns of parallel processing and synchronization. That is, it initiates the SobelEdges9 thread routine and ceiling(m/5) instances of the center slice thread routine. All of these instances run in parallel. The final statement waits for all threads to finish. To keep this example pedagogically simple, there is no logic in this design to handle exception cases such as one or more threads not completing. However, it is not hard to revise this DF into a new DF that does do that. This is left as an exercise for the reader.

(Speaker: Use the "Quick Peak at Synthetic Architecture" action button to go back to the synthetic Architecture slide and Point to each item in that slide that is referred to in the yellow expressions. That is, the edge expressions and their associated loop APCs when talking about "?DoOrderNCases", and the loop index name (i.e., "h") in the "Aslice" loop APC when talking about (Idex ?SlicerConstraint) .)



Framework: Instantiated Slicer

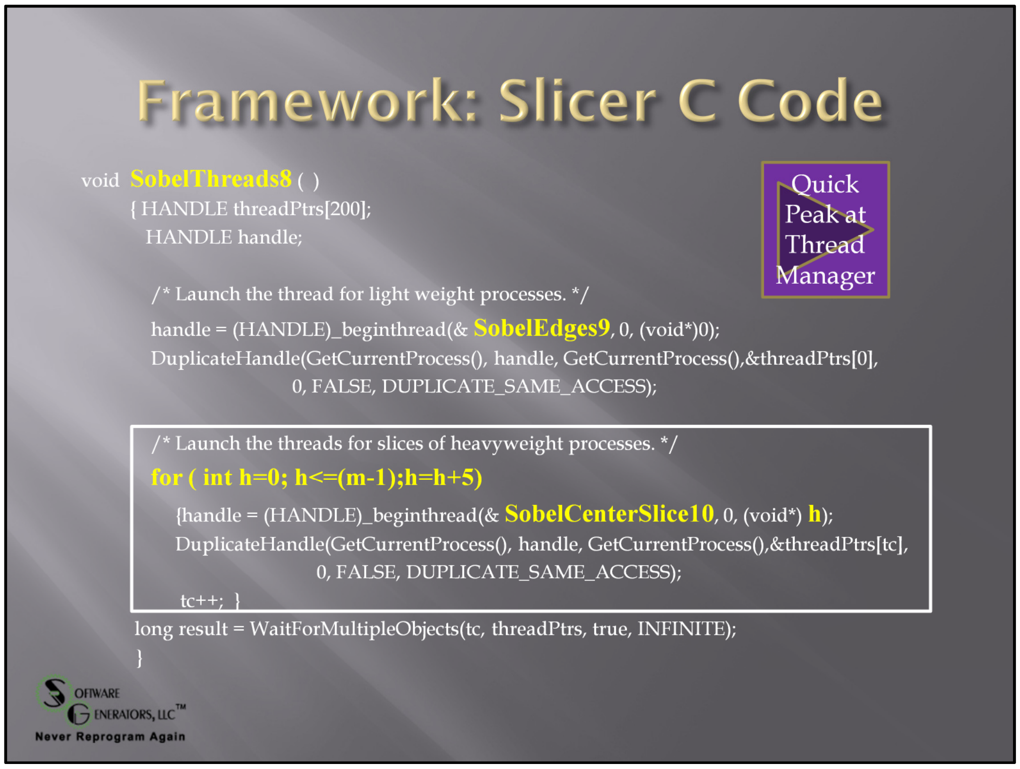
Quick Peak at Synthetic Architecture

```
void  SobelThreads8 ( )
      { HANDLE threadPtrs[200];
        HANDLE handle;

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

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

SOFTWARE GENERATORS, LLC™ Never Reprogram Again

So, this slide shows the results inlining of the bindings of the routine names and parameter name (which were introduced earlier) into the DF. That is, the simple inlining of **?managethreads**, **?DoOrderNCases** and **(Idex ?SlicerConstraint)**.

But we still have a loop APC "Slicer" that needs to be processed into a programming language loop. **… Next Slide …**

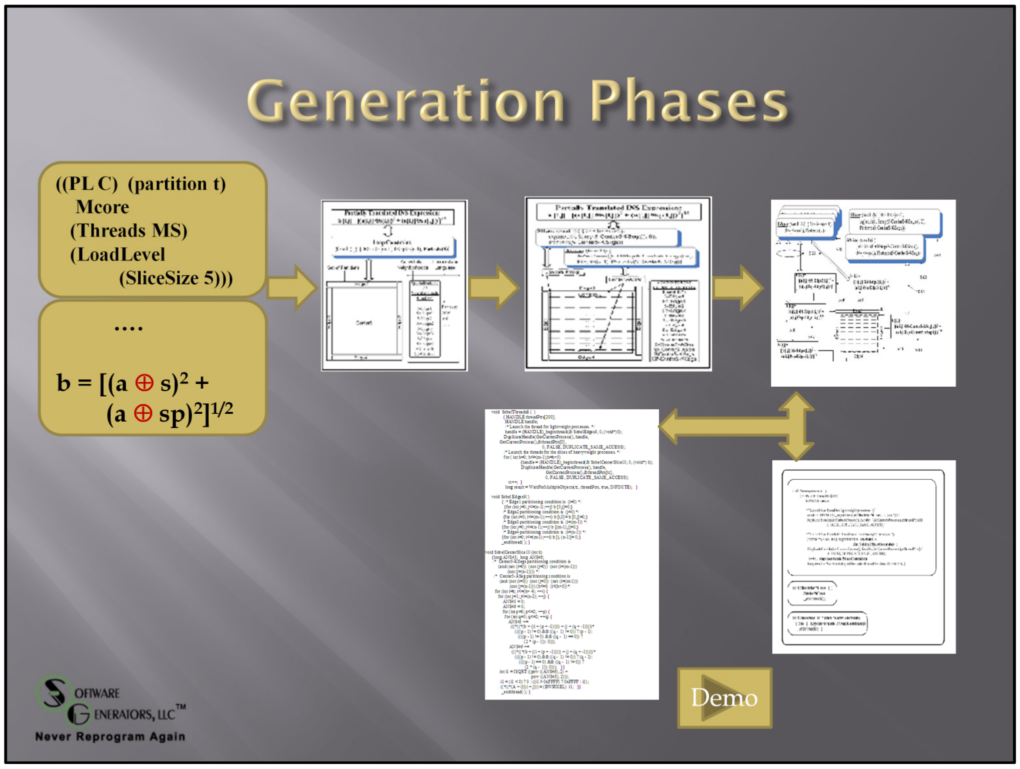


Framework: Slicer C Code

```
void  SobelThreads8 ( )
      { HANDLE threadPtrs[200];
        HANDLE handle;

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

         /* Launch the threads for 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);
       }
```

The loop generation process must create the loop over h from the logical conjuncts in the loop APC. The DF will use the Intermediate Language MT “Irange(Slicer)” to do this work, which will refine to code for the lower and upper limit of h. The loop increment comes from the Intermediate Language MT Rstep(S-Center5-Ksegs), which will refine to 5. The Rstep MT definition was created from the load-leveling specification in the user’s description of the execution platform.

At this point, DSLGen™ has generated the code that was used earlier in this presentation to illustrate the code generated for a multicore machine.



Generation Phases

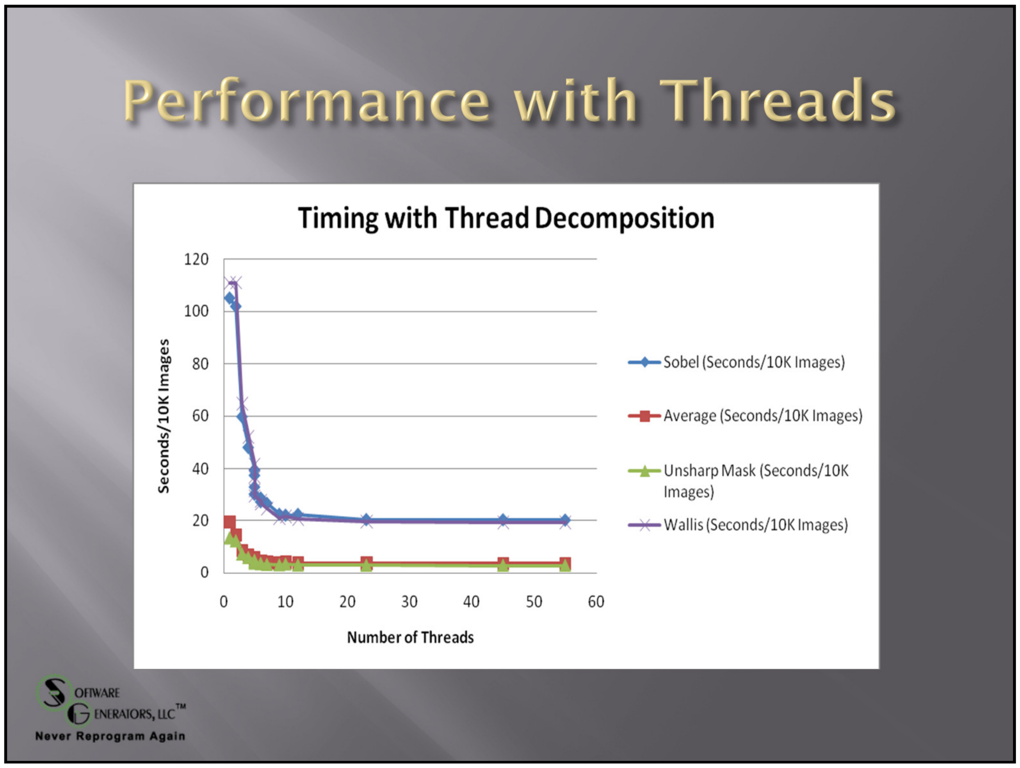
```
((PL C)  (partition t)
   Mcore
   (Threads MS)
   (LoadLevel
         (SliceSize 5)))
```

....

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

The final result is the Thread-based Slicer-Slicee code that we examined earlier in this presentation and that exploited the multicore architecture. This completes the example driven discussion of how DSLGen™ works.

If there is time and interest, the following (optional) slides discuss the performance of the generated code for various kinds of computations, various implementation architectures and various requirements imposed on the target implementation code.



Performance with Threads

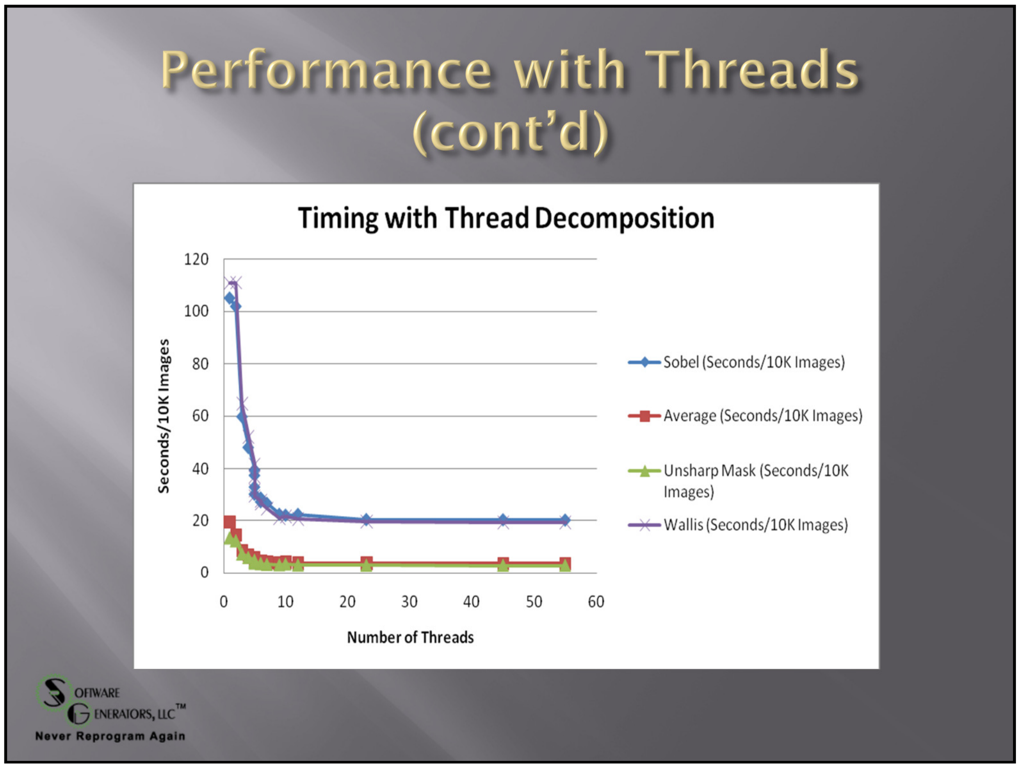
When “reading” the generated code, it was obvious that the various optimization strategies would improve performance but we had no numerical data to calibrate just how much. It was also clear that different kinds of computations would respond differently to the various performance improvement strategies. So, we did some timing tests to get a calibration of how effective the various optimization strategies actually were for the different kinds of computations. We chose four computations to test.

1. Sobel edge detection, which is computationally heavy because of the use of square and square root operations.
2. Wallis edge detection, which also was computationally heavy because it uses logarithms.
3. Image Average, which is a computationally lightweight using only addition and division but which illustrates an interesting design feature in the code – i.e., partition specific variations in the neighborhood loop control expressions.
4. Unsharp mask, another computational lightweight (it is used to sharpen mammogram images by clarifying the boundaries of tumor masses).

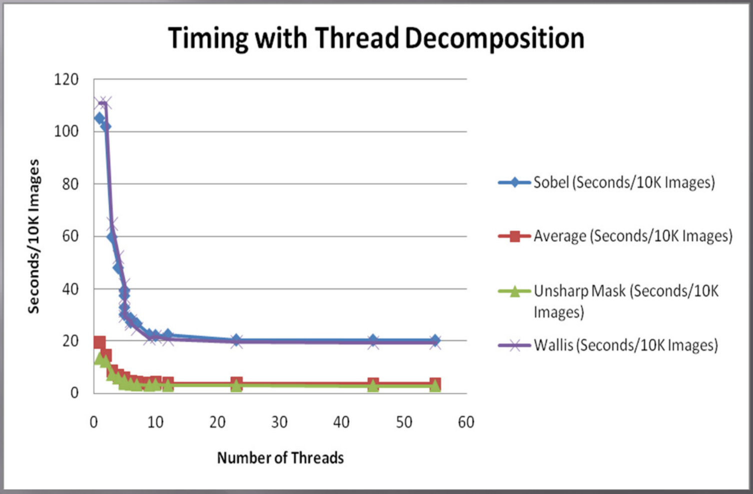
The generated code was tested on a 4 core, 3.33 GHz Velocity brand computer with 12 GB of real and 24 GB of virtual memory. The computer is built on the Intel i7 CPU with Turbo mode, which allows overclocking when the CPU is running under maximum temperature and power specification. It has 8 virtual processors. The code was compiled with Microsoft’s Visual Studio 2008 C/C++ compiler.

The test data was a 215 by 215 pixel image in RGB format with a 24 bit pixel depth.

(Continued on next slide….)



Performance with Threads (cont'd)



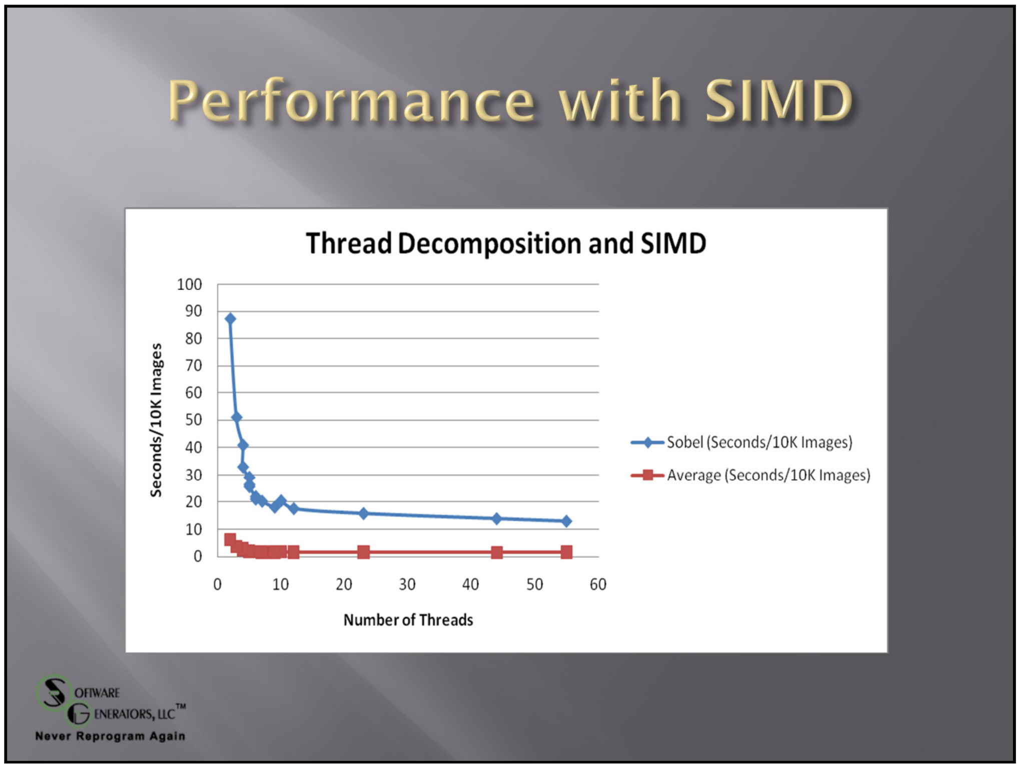
OFTWARE ENERATORS, LLC™
Never Reprogram Again

(Continued) This figure shows the results for various computations decomposed into threads to be run in parallel. These tests were run 10,000 times per image. For Sobel, the best performance was achieved at 55 threads, which required approximately 20.3 seconds to run the full set, or about 2 milliseconds per image. The worst results were with two threads, one for the edge cases and one for the center, which required approximately 105 seconds for the 10,000 images or a bit over a 10 millisecond per image. This was roughly the same time required for the calibration case, a hand coded version compiled to use no parallelism of any kind. Notice that the time drops quickly with five threads (i.e., one for the edges and four for the image center), taking about 32.8 seconds for the full set of images or about 3.3 milliseconds per image. This is about what simple logic would expect with four cores. However, the time continues to improve modestly for each five or so additional threads until it begins to level out at about 20.5 seconds at about 23 threads.

Interestingly enough this continued improvement appears to be a subtle effect with multicore parallelism for Sobel. One would expect little improvement by using significantly more threads than there are cores. That is, once you get beyond the number of real cores (i.e., 4 in this case), one would expect no further improvement by breaking the computation into more threads to be run in parallel. However, beyond 4 or 5 threads, the time continues to improve modestly for each five or so additional threads until it begins to level out at about 20.5 seconds at about 23 threads. Thereafter, the improvement is a tenth of a second or so for five or so additional threads. It is somewhat counter intuitive that one should get any improvement at all after the image has been evenly decomposed over the four cores. It is not entirely clear why this occurs but our current hypothesis is that it may be the "GPU effect" where many threads can mask memory, cache or other kind of latency if thread switching is efficient enough. Also, fast thread switching among virtual processors in the hardware (called Hyperthreading) may play a role. The target computer has two virtual processors per core and this is known to increase overall performance in many cases.

We get further improvement with the addition of SIMD (vector) instructions. The following slide shows the results for one heavyweight computation (Sobel) and one lightweight computation (Image Average).

(See C:\Users\Public\Documents\Graphics\Documentation\Performance Data\[Subset of Actual Performance Numbers - 5-2-2010 For Documents.xlsx]Sheet1)

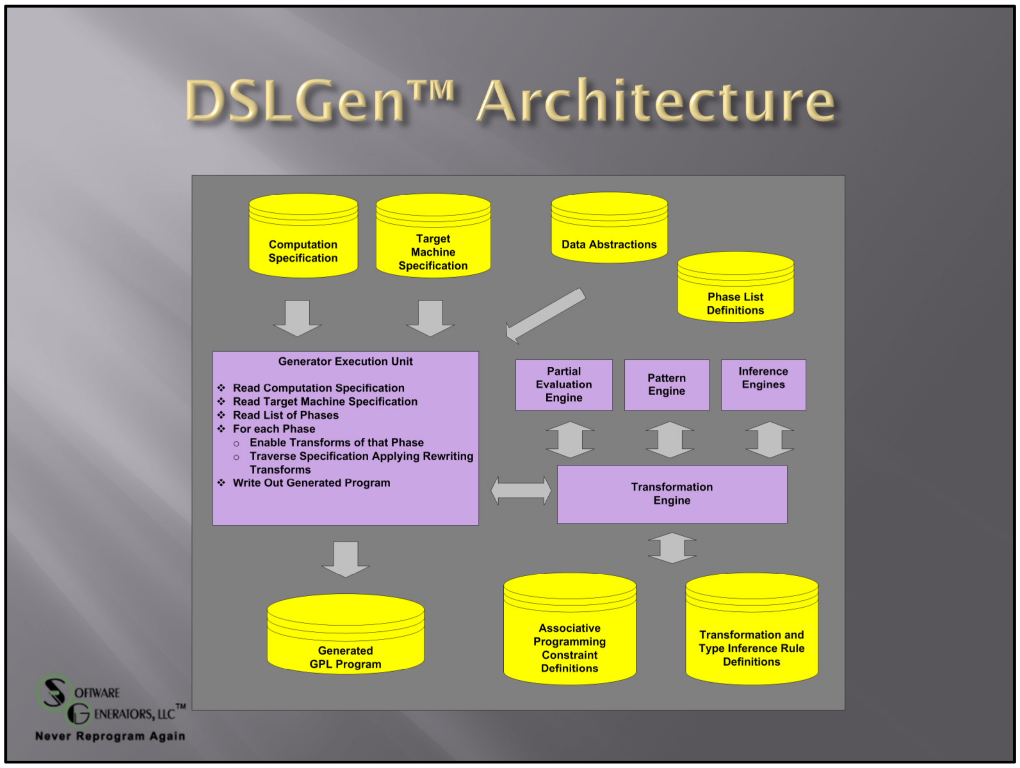


Performance with SIMD

With the addition of SIMD instructions, the added improvement for Sobel ranges from about a 14% improvement for few or no threads to 36% for the maximum number of threads tested. With only two threads, Sobel took 87.2 seconds for all 10K images or 8.7 milliseconds per image, whereas with 55 threads, it took 12.87 seconds for all 10K images or about 1.3 milliseconds per image.

Interestingly, the effect for Image Average was significant and somewhat surprising – between 57% and 58% improvement regardless of the number of threads. What this suggests is that virtually all of the computation for Image Average can be done with instruction level parallelism (i.e., addition of a vector of numbers) leaving only a single additional arithmetic operation (i.e., multiplication or division by a constant) to be done via the standard arithmetic unit. So, the multicore parallelism improvement is a much flatter curve. Once one runs out of processors there is little additional effect although there may be some additional effect depending on the structure of the computation and its possible effect of processor latencies. Recall the comments from the previous slide about the counter intuitive improvement for Sobel beyond 5 or so threads.

By contrast, a much smaller amount of the Sobel computation can be done in parallel instructions (i.e., the PMADD operation on the three weight vectors and the three corresponding pixel vectors, where each of the vectors is only three numbers long) whereas the addition of the intermediate instructions (via the PADD instruction) is relatively inconsequential. But more importantly, the square, addition and square root operations that follow the SIMD instructions will be done on the standard CPU arithmetic unit. These last three operations (and especially the square root operation) are likely to be the lion's portion of the computation. Therefore, the really big improvement with Sobel arises because of the large scale parallelism provided by the thread based parallelism and this tends to swamp much of the savings of the instruction level parallelism of the SIMD instructions.



DSLGen™ Architecture

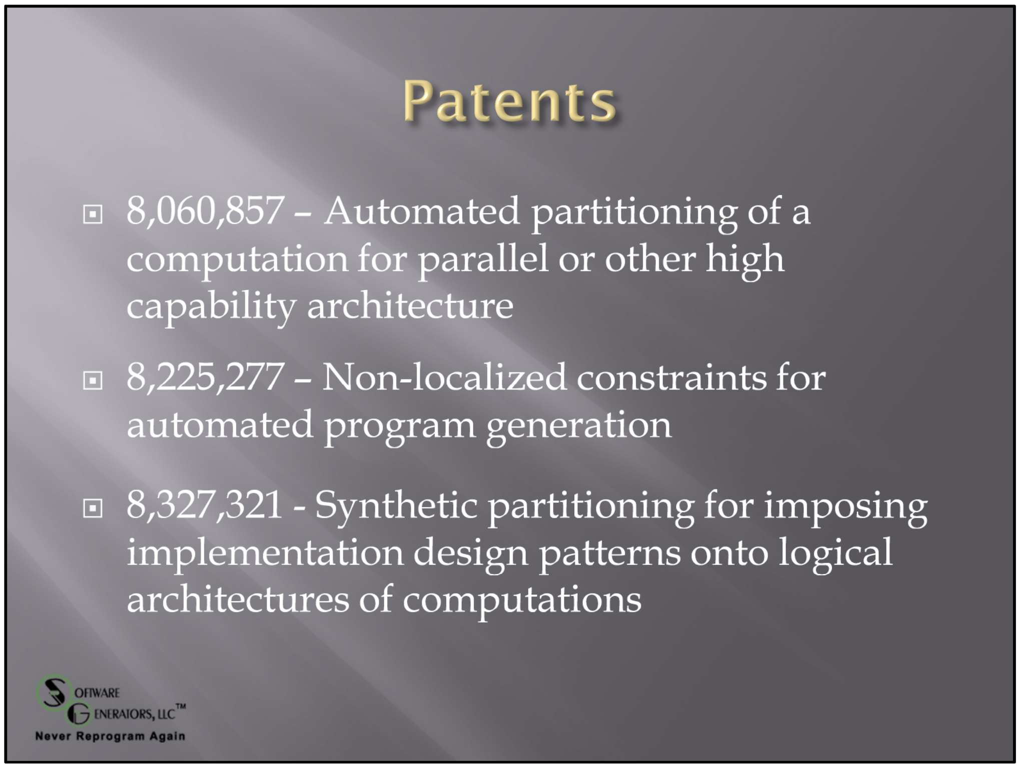
Generator Execution Unit

- Read Computation Specification
- Read Target Machine Specification
- Read List of Phases
- For each Phase
  - Enable Transforms of that Phase
  - Traverse Specification Applying Rewriting Transforms
- Write Out Generated Program

Computation Specification
Target Machine Specification
Data Abstractions
Phase List Definitions
Partial Evaluation Engine
Pattern Engine
Inference Engines
Transformation Engine
Generated GPL Program
Associative Programming Constraint Definitions
Transformation and Type Inference Rule Definitions
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

The broad component architecture of DSLGen™ is shown in this figure. The architecture is "microprogrammable" in the sense that virtually all aspects of DSLGen™ are data defined. This allows a domain engineer to create custom generators for arbitrary DSLs, to vary old or introduce new execution platform targets and to add definitions for new output languages or to tweak the definitions of an existing language to fit a custom variant of that language.

The foregoing has touched upon various components of DSLGen™ such as the Partial Evaluation (PE) engine without a great deal of operational depth. However, the supplementary Tools Demo presentation will show some of these components in operation within a specific problem context. For example, it will reveal how the initial Logical Architecture (LA) is formed via an "animation", the role of the PE engine in the simplification of the edge processing loops, the operation of the History debugger, and so forth.



Patents

- 8,060,857 – Automated partitioning of a computation for parallel or other high capability architecture
- 8,225,277 – Non-localized constraints for automated program generation
- 8,327,321 - Synthetic partitioning for imposing implementation design patterns onto logical architectures of computations

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Three patents that have been issued on this R&D so far.

SOFTWARE GENERATORS, LLC™

Never Reprogram Again

SOFTWARE GENERATORS, LLC™

Never Reprogram Again