The SPARAMAT Approach to Automatic Comprehension of Sparse Matrix Computations*

Christoph W. Keßler Helmut Seidl Craig H. Smith

Fachbereich IV - Informatik, Universität Trier, 54286 Trier, Germany
e-mail: {kessler,smith}@psi.uni-trier.de

Abstract

Automatic program comprehension is particularly useful when applied to sparse matrix codes*, since it allows to abstract e.g. from specific sparse matrix storage formats used in the code. In this paper we describe SPARAMAT, a system for speculative automatic program comprehension suitable for sparse matrix codes, and its implementation.*

1 Introduction

Matrix computations constitute the core of many scientific numerical programs. A matrix is called *sparse* if so many of its entries are zero that it seems worthwhile to use a more space–efficient data structure to store it than a simple two–dimensional array; otherwise the matrix is called *dense*. Space–efficient data structures for sparse matrices try to store only the nonzero elements. This results in considerable savings in space for the matrix elements and time for operations on them, at the cost of some space and time overhead to keep the data structure consistent. If the spatial arrangement of the nonzero matrix elements (the *sparsity pattern*) is statically known to be regular (e.g., a blocked or band matrix), the matrix is typically stored in a way directly following this sparsity pattern; e.g., each diagonal may be stored as a one–dimensional array.

Irregular sparsity patterns are usually defined by runtime data. Here we have only this case in mind when using the term "sparse matrix". Typical data structures used for the representation of sparse matrices in Fortran77 programs are, beyond a *data array* containing the nonzero elements themselves, several *organizational variables*, e.g. arrays with suitable row and/or column index information for each data array element. Linked lists are, if at all, simulated by index vectors, as Fortran77 supports no pointers nor structures. C implementations may also use explicit linked list data structures to store the nonzero elements, which supports dynamic insertion and deletion of elements. However, on several architectures, a pointer variable needs more space than an integer index variable. As space is often critical in sparse matrix computations, explicit linked lists occur rather rarely in practice. Also, many numerical C programs are written in a near–Fortran77 style because they were either directly transposed from existing Fortran77 code, or because the programming style is influenced by former Fortran77 projects or Fortran77–based numerics textbooks.

Matrix computations on these data structures are common in practice and often parallelizable. Consequently, numerous parallel algorithms have been invented or adapted for sparse matrix computations over the last decades for various parallel architectures.

[5] suggests the programmer to express, in the source code, parallel (sparse) matrix computations in terms of dense matrix data structures, which are more elegant to parallelize and distribute, and let the compiler select a suitable data structure for the matrices automatically. Clearly this is not applicable to (existing) programs that use hard–coded data structures for sparse matrices.

While the problems of automatic parallelization for *dense* matrix computations are, meanwhile, well understood and sufficiently solved, (e.g. [6, 24, 46]), these problems have been attacked for *sparse* matrix computations only in a very conservative way, e.g., by run–time parallelization techniques such as the inspector–executor method [32] or run–time analysis of sparsity patterns for load–balanced array distribution [45]. This is not astonishing because such code looks quite awful to the compiler, consisting of indirect array indexing or pointer dereferencing which makes exact static access analysis impossible.

In this paper we describe SPARAMAT, a system for concept comprehension that is particularly suitable to *sparse* matrix codes. We started by studying several representative source codes for implementations of basic linear algebra operations like dot product, matrix–vector multiplication, matrix–matrix multiplication, or LU factorization for sparse matrices [17, 20, 29, 43, 41, 48] and recorded a list of basic computational kernels for sparse matrix computations, together with their frequently occurring syntactical and algorithmic variations.

Basic terminology. A *concept* is an abstraction of an externally defined procedure. It represents the (generally infinite) set of concrete procedures coded in a given programming language that have the same type and that we consider to be equivalent in all occurring calling contexts. Typically we give a concept a *name* that we associate with the type and the operation that we consider to be implemented by these procedures. An *idiom* of a concept c is such a concrete

*Research partially funded by DFG, project SPARAMAT

procedure, coded in a specific programming language, that has the same type as c and that we consider to implement the operation of c. An *occurrence* of an idiom i of a concept c (or short: an occurrence of c) in a given source program is a fragment of the source program that matches this idiom i by unification of program variables with the procedure parameters of i. Thus it is legal to replace this fragment by a call to c, where the program objects are bound to the formal parameters of c. The (compiler) data structure representing this call is called an *instance* I of c; the fields in I that hold the program objects passed as parameters to c are called the *slots* of I. Beyond the Fortran77 parameter passing, SPARAMAT allows procedure–valued parameters as well as higher–dimensional and composite data structures to occur as slot entries.

After suitable preprocessing transformations (inlining all procedures) and normalizations (constant propagation), the intermediate program representation — abstract syntax tree and/or program dependence graph — is submitted to the concept recognizer. The concept recognizer, described in Section 4, identifies code fragments as concept occurrences and annotates them by concept instances.

When applied to parallelization, we are primarily interested in recognizing concepts for which there are particular parallel routines available, tailored to the target machine. In the back–end phase, the concept instances can be replaced by suitable parallel implementations. The information derived in the recognition phase also supports automatic data layout and performance prediction.

Problems with sparse matrix computations. One problem we were faced with is that there is no standard data structure to store a sparse matrix. Rather, there is a set of about 15 competing formats in use that vary in their advantages and disadvantages, in comparison to the two–dimensional array which is the "natural" storage scheme for a dense matrix.

The other main difference is that space–efficient data structures for sparse matrices use either indirect array references or (if available) pointer data structures. Thus the array access information required for safe concept recognition and code replacement is no longer completely available at compile time. Regarding program comprehension, this means that it is no longer sufficient to consider only the declaration of the matrix and the code of the computation itself, in order to safely determine the semantics of the computation. Code can only be recognized as an occurrence of, say, sparse matrix–vector multiplication, subject to the condition that the data structures occurring in the code really implement a sparse matrix. As it is generally not possible to statically evaluate this condition, a concept recognition engine can only *suspect*, based on its observations of the code while tracking the live ranges of program objects, that a certain set of program objects implements a sparse matrix; the final *proof* of this hypothesis must either be supplied by the user in an interactive program understanding framework, or equivalent run-time tests must be generated by the code generator. Unfortunately, such run–time tests, even if parallelizable, incur some overhead. Nevertheless, static program flow analysis [25] can substantially support such a *speculative* comprehension and parallelization. Only at program points where insufficient static information is available, run–time tests or user prompting is required to confirm (or reject) the speculative comprehension.

Application areas. The expected benefit from successful recognition is large. For automatic parallelization, the back-end should generate two variants of parallel code for the recognized program fragments: (1) an optimized parallel library routine that is executed speculatively, and (2) a conservative parallelization, maybe using the inspector–executor technique [32], or just sequential code, which is executed non–speculatively. These two code variants may even be executed concurrently and overlapped with the evaluation of run-time tests: If the testing processors find out during execution that the hypothesis allowing speculative execution was wrong, they abort and wait for the sequential variant to complete. Otherwise, they abort the sequential variant and return the computed results. Nevertheless, if the sparsity pattern is static, it may be more profitable to execute the run-time test once at the beginning and then branching to the suitable code variant.

Beyond automatic parallelization, the abstraction from specific data structures for the sparse matrices also supports program maintenance and debugging, and could help with the exchange of one data structure for a sparse matrix against another, more suitable one. For instance, recognized operations on sparse matrices could be replaced by their counterparts on dense matrices, and thus, program comprehension may serve as a front end to [5]. Or, the information derived by concept recognition may just be emitted as mathematical formulas e.g. in LaTeX format, typeset in a mathematical textbook style, and shown in a graphical editor as annotations to the source code, in order to improve human program understanding.

The SPARAMAT implementation focuses on sparse matrix computations coded by indirect array accesses. This is because, in order to maintain an achievable goal in a university project, it is necessary to limit oneself to a language that is rather easy to analyze (Fortran), to only a handful of sparse matrix formats (see Section 2), and to a limited set of most important concepts [26]. For this reason, pointer alias analysis of C programs, as well as concepts and matching rules for pointer–based linked list data structures, are beyond the scope of this project. Due to the flexibility of the generative approach, more concepts and templates may be easily added by any SPARAMAT user. Furthermore, it appears that we can reuse some techniques from our earlier PARAMAT project [24] more straightforwardly for indirect array accesses than for pointer accesses.

The remainder of this paper is organized as follows: Section 2 deals with vectors and sparse matrix storage schemes; Section 3 summarizes concepts for (sparse) matrix computations. Section 4 discusses concept recognition and describes our implementation. We close with related work and conclusions. A larger example using neural network simulation code is given in Appendix A.

2 Vectors and (sparse) matrices

2.1 Vectors

A *vector* is an object in the intermediate program representation that summarizes a one–dimensional view of some elements of an array. For instance, a vector of reals accessing the first 5 elements in column 7 of a two–dimensional array `a` of reals is represented as `V(a,1,5,1,7,7,0)`. For ease of notation we assume that the "elements" of the vector itself are consecutively numbered starting at 1. `IV(...)` denotes integer vectors.

An *indexed vector* summarizes a one–dimensional view of some elements of an array whose indices are specified in a second (integer) vector, e.g. `VX(a,IV(x,1,n,2))`.

2.2 (Sparse) Matrices

A *matrix* summarizes a two–dimensional view of an array according to the conventions of a specific storage format. Dense matrices appear as a special case of sparse matrices.

Here we summarize some general storage formats for sparse matrices based on index vectors, which are the most frequently occurring in Fortran77 codes. Formats for special, more regular sparsity patterns, such as for band matrices, block sparse matrices, or skyline matrices, are not considered here. The abbreviations of format names are partially adapted from [41]. More details can be found in [41], [2], and [47].

- **DNS** (*dense storage format*): uses a two–dimensional array `A(N,M)` to store all elements. Due to the symmetric access structure of the two–dimensional array, a *leading dimension* flag `ld` tells us whether the matrix is stored row–major or column–major. In the following, we summarize all data referring to the dense matrix as an object

```
DNS( V(a,1,n,1,m), n, m, ld )
```

  Example: In Fortran, DNS–Matrix–vector multiplication may look like

```
DO i = 1, n
   b(i) = 0.0
   DO j = 1, m
      b(i) = b(i) + a(i,j) * x(j)
   ENDDO
ENDDO
```

- **COO** (*coordinate format*): A data array `a(nz)` stores the `nz` nonzero matrix elements in arbitrary order, and integer vectors `row(nz)` and `col(nz)` hold for each nonzero element its row and column index. The object representing the matrix is summarized as

```
COO( V(a,1,nz,1), IV(row,1,nz,1), IV(col,1,nz,1),
     nz )
```

  Example: COO–Matrix–vector multiplication may look like

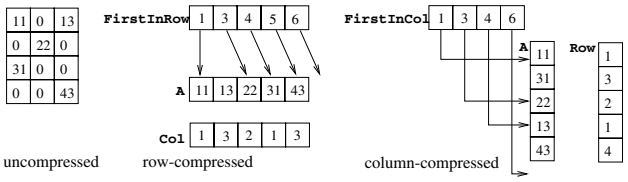

Figure 1: Row–compressed (CSR) and column–compressed (CSC) storage formats for sparse matrices.

```
DO i = 1, n
   b(i) = 0.0
ENDDO
DO k = 1, nz
   b(row(k)) = b(row(k)) + a(k) * x(col(k))
ENDDO
```

  The COO format occurs e.g. in the SLAP package [43].

- **CSR** (*row–compressed sorted storage format*): A data array `a(nz)` stores the `nz` nonzero matrix elements a_{ij} in row–major order, where within each row the elements appear in the same order as in the dense equivalent. An integer vector `col(1:nz)` gives the column index for each element in `a`, and an integer vector `firstinrow(1:n+1)` gives indices to `a` such that `firstinrow(`i`)` denotes the position in `a` where row i starts, $i = 1,...,$`n` and `firstinrow(n+1)` always contains `nz+1` (see Figure 1). Thus, `firstinrow(`i`+1)-firstinrow(`i`)` gives the number of nonzero elements in row i. A CSR matrix object is summarized as

```
CSR(V(a,firstinrow(1),firstinrow(n+1)-1,1),
    IV(firstinrow,1,n+1,1),
    IV(col,firstinrow(1),firstinrow(n+1)-1,1),
    n,
    nz)
```

  Example: An idiom of a matrix vector multication for CSR format may look like

```
DO i = 1, n
   b(i) = 0.0
   DO k = firstinrow(i), firstinrow(i+1)-1
      b(i) = b(i) + a(k) * x(col(k))
   ENDDO
ENDDO
```

  Such storage formats are typical for Fortran77 implementations. CSR is used e.g. in the SLAP package [43].

- **CUR** (*row–compressed unsorted storage format*): like CSR, but the order of nonzeros within each row is not important. CUR is used e.g. as the basic format in SPARSKIT [41]. CUR matrix–vector multiplication looks identical to the CSR version.

- **XSR / XUR**: an extension of CSR / CUR that additionally stores in an n–element integer array `lastinrow`

for each compressed row its last index within the data array A. This makes row interchanges and row reallocations due to fill–in more efficient. XUR is used e.g. in Y12M [48].

- **MSR** (*modified row–compressed storage format*): like CSR, but the elements of the main diagonal of the matrix are stored separately and regardless of whether they are zero or not. This is motivated by the fact that, often, most of the diagonal elements are a priori known to be nonzero, and are accessed more frequently than the other elements. Typically the diagonal elements are stored in the first n elements of a and a(n+1) is unused. The column indices of the diagonal elements need not be stored, thus the elements of the array firstinrow of CSR are stored in the first n+1 entries of a two–purpose integer array fircol. The remaining nonzero elements are stored in a(n+2:nz+1) and their column indices in fircol(n+2:nz+1). A MSR matrix object is thus given as

```
MSR(V(a,1,fircol(n+1)-1,1), IV(fircol,1,n+1,1), n,
    nz)
```

  MSR is used e.g. in the sparse matrix routines of the *Numerical Recipes* [37].

  Example: Matrix–vector multiplication may look as follows (routine sprsax() from [37]):

```
DO i = 1, n
    b(i) = a(i) * x(i);
    DO k = fircol(i), fircol(i+1)-1
       b(i) = b(i) + a(k) * x(fircol(k))
    ENDDO
ENDDO
```

- **CSC** (*column–compressed format*): similar to CSR where the a contains the nonzero elements in column–major order and the other two arrays are defined correspondingly (see Figure 1). Thus, CSC format for a matrix A is equivalent to the CSR format for A^T, and vice versa. A CSC matrix object is summarized by

```
CSC(V(a,firstincol(1),firstincol(n+1)-1,1),
    IV(firstincol,1,n+1,1),
    V(row,firstincol(1),firstincol(n+1)-1,1),
    n, nz)
```

  Example: CSC–Matrix–vector multiplication may look like

```
DO i = 1, n
   DO k = firstincol(i), firstincol(i+1)-1
      b(row(k)) = b(row(k)) + a(k) * x(i)
   ENDDO
ENDDO
```

  CSC is used e.g. in the Harwell MA28 package [17].

- **MSC** (*modified column–compressed storage format*): A MSC matrix object is similar to the CSC representation, but the elements of the main diagonal of the matrix are stored separately, as for MSR.

- **JAD** (*jagged diagonal format*): First the rows of the matrix are permuted to obtain decreasing numbers n_i of nonzero elements for each row i. The data array a(1:nz) is filled as follows: The first nonzero element of each row i (the first "jagged diagonal") is stored in a(i), the second nonzero element of each row i in a(n+i) etc. The overall number njd of jagged diagonals is at most n. An integer array col(1:nz) holds the column index of each element in a. An integer array firstinjdiag(1:n) holds indices into a resp. col indicating the beginning of a new jagged diagonal; thus firstinjdiag($k+1$)–firstinjdiag(k) gives the number n_k of elements belonging to the kth jagged diagonal. Thus, a JAD matrix object is given by

```
JAD( V(a,firstinjdiag(1), firstinjdiag(njd+1),1),
     IV(firstinjdiag,1,njd+1,1),
     V(col,firstinjdiag(1),firstinjdiag(njd+1)-
1,1),
     n, nz, njd )
```

  Example: JAD–Matrix–vector multiplication may look like

```
DO r = 1, n
   b(r) = 0.0
ENDDO
DO k = 1, njd
   DO i = firstinjdiag(k), firstinjdiag(k+1)-1
      r = i - firstinjdiag(k)
      b(r) = b(r) + a(i) * x(col(i))
   ENDDO
ENDDO
```

  followed by re–permutation of vector b if necessary.

- **LNK** (*linked list storage format*): The data array a(1:maxnz) holds the nz nonzero elements in arbitrary order, the integer array col(1:maxnz) gives the column index of each nonzero element. An integer array nextinrow(1:maxnz) links the elements belonging to the same row in order of increasing col index. A zero nextinrow entry marks the last nonzero element in a row. The list head element of each row i is indexed by the ith element of the integer array firstinrow(1:n). Empty rows are denoted by a zero firstinrow entry. If required by the application, a similar linking may also be provided in the other dimension, using two more index vectors nextincol(1:nz) and firstincol(1:n). Thus, a singly–linked LNK matrix object is summarized by

```
LNK(VX(a,IV(firstinrow,1,n)),IV(firstinrow,1,n),
    IV(nextinrow,1,n),
    VX(col,IV(firstinrow,1,n)),n,nz,maxnz)
```

  Example: LNK–Matrix–vector multiplication may look like

```
DO i = 1, n
   b(i)=0.0
   k = firstinrow(i)
   WHILE (k.GT.0)
      b(i) = b(i) + a(k) * x(col(k))
      k = nextinrow(k)
   ENDWHILE
ENDDO
```

The LNK format requires more space than the previously discussed sparse matrix formats, but it supports efficient dynamic insertion and deletion of elements (provided that `a` and `nextinrow` have been allocated with sufficient space reserve, `maxnz`).

While matrix–vector multiplication codes for a sparse matrix look quite simple and seem to be somehow identifiable by concept matching techniques, implementations of matrix–matrix multiplication or LU decomposition look quite unstructured. This is mainly due to the fact that in the course of these algorithms, some matrix elements may become nonzero which were originally zero (*fill–in*), and thus additional storage has to be allocated for inserting them. Thus, the sparsity pattern may change in each step of these algorithms, while at matrix–vector multiplication, the sparsity pattern (and thus, the organizational variables) is read–only.

A simple work–around to cope with a limited number of fill–ins is to store fill–ins in a separate temporary data structure, or respectively, to allocate slightly more space for the data array and the index vectors. This is e.g. applied in `SPARSE` [29].

There are also many possibilities for slight modifications and extensions of these data structures. For instance, a flag may indicate symmetry of a matrix. Such changes are quite ad–hoc, and it seems generally not sensible to define a new family of concepts for each such modification. For instance, in the Harwell routines `MA30`, the sign bit of the row resp. column indices is "misused" to indicate whether a new column or row has just started, thus saving the `firstinrow` resp. `firstincol` array when sequentially scanning through the matrix. Clearly such dirty tricks make program comprehension more difficult.

A main consequence that arises from these data structures is that the comfortable symmetry present in the two–dimensional arrays implementing dense matrices (DNS) is lost. Hence, we must explicitly distinguish between transposed and non–transposed matrix accesses, and between row–wise and column–wise linearization of the storage for the nonzero matrix elements.

Linked list data structures (e.g., the LNK format) cause operations on them, such as traversal or insert/delete, to be inherently sequential. Thus they are particularly good candidates to be completely replaced by other data structures more suitable for exploiting parallelism, e.g. linked lists with multiple heads for parallel access. Data structure replacement for a sparse matrix is possible if all operations on it have been recognized and if alias analysis can guarantee that there are no other variables which may be used to access one of these linked list elements in an unforeseen way.

3 Concepts

This section gives a survey of concepts that are frequently encountered in sparse matrix codes. Although this list is surely not exhaustive, it should at least illustrate the application domain. The extension of this list by more concepts to cover an even larger part of numerical software is the subject of on-going research.

```
concept SDOTVV {
  param(out)  $r: real;
  param(none) $L: range;
  param(in)   $u: vector;
  param(in)   $v: vector;
  param(in)   $init: real;

  templateVertical {
    pattern {
       node  DO_STMT $i = $lb:$ub:$st
       child    INCR($rs,MUL($e1,$e2))
    }
    where {
          $e1->isSimpleArrayAccess($i)
          && $e2->isSimpleArrayAccess($i)
          && $s->isVar()
          && $i->notOccurIn($s)
    }
    instance SDOTVV($rs, newRange($i,$lb,$ub,$st),
                    newVector($e1,$i,$lb,$ub,$st),
                    newVector($e2,$i,$lb,$ub,$st),
                    $rs)
  }
  templateHorizontal {
    pattern {
      sibling($s) SINIT($x,$c)
         fill($f)
         node($n) SDOTVV($r1,$L1,$u1,$v1,$init1)
    }
    where($s) { $x->array() == $init1->array() }
    where($f) {
      notOutSet = $x;
      notInSet  = $x;
      inSet     = $init1;
    }
    instance($s) EMPTY()
    instance($n) SDOTVV( $L1, $u1, $v1, $r1, $c )
  }
}
```

Figure 2: A CSL specification for the `SDOTVV` concept (simple dot product) with two templates.

We have developed a concept specification language that allows one to describe concepts and matching rules on a level that is (more or less) independent from a particular source language or compiler. A concept specification consists of the following components: its name (naming conventions are discussed below), an ordered and typed list of its parameters, and a set of matching rules (called *templates*). A matching rule has several fields: a field for structural pattern matching, specified in terms of intermediate representation constructs (loop headers, conditions, assignments, and instances of the corresponding subconcepts), fields specifying auxiliary predicates (e.g., structural properties or dataflow relations), fields for the specification of pre- and postconditions for the slot entries implied by this concept (see Section 4), and a field creating a concept instance after successful matching. For an example specification see Figure 2.

Our naming conventions for concepts are as follows: The shape of operands is denoted by shorthands `S` (scalar), `V` (vector), `VX` (indexed vector), and `YYY` (matrix in storage format YYY). The result shape is given first, followed by a mnemonic for the type of computation denoted by the concept, and the shorthands of the operands. The default type is real; integer concepts and objects are prefixed with an `I`.

We extend our earlier approach [24] to representing concepts and concept instances in several aspects.

Operator parameters. Some concepts like VMAPVV (elementwise application of a binary operator to two operand vectors) take an operator (i.e., a function pointer) as a parameter. This makes hierarchical program omprehension slightly more complicated, but greatly reduces the number of different concepts, and allows for a more lean code generation interface.

Functional composition. We are still discussing arbitrary functional composition of concepts to form new concepts. This idea is inspired by the work of Cole on algorithmic skeletons [13]. Nevertheless, there should remain at least some "flat" concepts for important special cases, e.g. SDOTVV for dot product, VMATVECMV for matrix–vector multiplication, etc. These may be regarded as "syntactic sugar" but are to be preferred as they enhance readability and speed up the program comprehension process.

No in–place computations. Most of our concepts represent not–in–place computations. In general, recognized in–place computations are represented by using temporary variables, vectors, or matrices. This abstracts even further from the particular implementation. It is the job of the backend to reuse (temporary array) space where possible. In other words, we try to track *values* of objects rather than memory locations. Where it is unavoidable to have accumulating concepts, they can be specified using accumulative basic operations like INCR (increment) or SCAL (scaling).

Concept instances as parameters. Nesting of concept instances is a natural way to represent a tree–like computation without having to specify temporary variables. As an example, we may denote a DAXPY–like computation as

```
VMAPVS( V(tmp,1,n,1), MUL, V(c,1,n,1), 3.14 )
VINCRV( V(b,1,n,1), V(tmp,1,n,1) )
```

which is closer to the internal representation in the compiler, or as

```
VINCR( V(b,1,n,1), VMAPVS(MUL,V(c,1,n,1),3.14))
```

which is more readable for humans. If the computation structure is a directed acyclic graph (DAG), then we may also obtain a DAG of concept instances, using temporary variables and arrays for values used multiple times. In order to support nesting, our notation of concept instances allows to have the result parameter (if there is exactly one) of a concept instance appear as the "return value" of a concept instance, rather than as its first parameter, following the analogy to a call to a function returning a value.

We give here an informal description of some concepts. v, v_1, v_2 denote (real) vectors, a a real array, iv an integer vector, m, m_1, m_2 matrices in some format and r a range object. i, i_1,...,i_5 denote integer valued concept instances.

3.1 Base Concepts

V$(a, l_j, u_j, s_j, ...)$	real vector access of array a, where l_j, u_j, s_j are lower, upper and stride of dimension j where j varies from 1 to the maximum number of dimensions.
IV$(a, l_1, u_1, s_1, ...)$	integer vector access of array a, where l_j, u_j, s_j are lower, upper and stride of dimension j where j varies from 1 to the maximum number of dimensions.
VX$(i, IV(v_1, ...))$	real indirect access of array i by array v_1.
IVX$(i, IV(v_1...))$	integer indirect access of array i by array v_1.
VAR$(v, i_1, ..., i_5)$	real variable access up to five dimensions. If no indices specified, then scalar access.
IVAR$(v, i_1, ..., i_5)$	integer variable access up to five dimensions. If no indices specified, then scalar access.
CON(c)	real constant c
ICON(c)	integer constant c
VCON(r, n)	vector constant size n containing real r
IVCON(i, n)	vector constant size n containing integer i
EMPTY$()$	no operation
RANGE(lv, lb, ub, st)	lv is the ranging variable and lb, ub, st are the lower bound, upper bound and stride

For sake of brevity, VAR and IVAR concept instances are written as their original expressions.

3.2 Concepts for scalar computations

There are concepts for binary expression operators, like ADD, MUL, MAX, EQ etc., for unary expression operators like NEG (negation), ABS (absolute value), INV (reciprocal), SQR (squaring) etc., The commutative and associative operators, ADD, MUL, MAX etc., MIN, OR, AND may also have more than two operands. STAR is a special version of a multi–operand ADD denoting difference stencils [24]. The increment operators INCR (for accumulating addition) and SCAL (for accumulating product) are used instead of ADD or MUL where the result variable is identical to one of the arguments. Assignments to scalars are either SCOPY where the assignee is a variable, or SINIT where the assignee is a constant, or an expression operator where the assignee is a recognized expression. The default type is real; integer versions of these concepts are prefixed with an I. For technical reasons there are some auxiliary concepts like EMPTY (no operation) and RANGE (to summarize a loop header).

3.3 Vector and matrix computations

VMAPVV$(v, \oplus, v_1, v_2)$	elementwise appl. of binary operator $\oplus$ results stored in v
VMAPV$(v, \ominus, v_1)$	elementwise appl. of unary operator $\ominus$ results stored in v
VMAPVS$(v, \oplus, v_1, r)$	elementwise appl. with a scalar operand r results stored in v
VINCRV(v, v_1)	$v(i) = v(i) + v_1(i),\ i = 1, ..., \lvert v_1 \rvert$
VCOPYV(v, v_1)	copy vector v_1 to v
VINIT(v, c)	initialize elements of v to a constant c
VASGNS(v, r)	initialize elements vector v to a scalar r
SREDV$(r, \otimes, v)$	reduction $r = \bigotimes_{j=1}^{\lvert v \rvert} v(j)$
SREDLOCV$(k, \odot, v)$	some k with $v(k) = \bigodot_{j=1}^{\lvert v \rvert} v(j)$
VPREFV$(v, \otimes, v_1)$	$v(i) = \bigotimes_{j=1}^{i} v_1(j), i = 1, ..., \lvert v_1 \rvert$
VSUFFV$(v, \otimes, v_1)$	$v(i) = \bigotimes_{j=\lvert v_1 \rvert}^{i} v_1(j), i = \lvert v_1 \rvert, ..., 1$

3.4 Searching and sorting on a vector

SSRCHV(k, v_1, r)	k = rank of r in v_1
VSORTV(v, v_1)	sort v_1 and store result in v
VCOLLVV(v, v_1, v_2)	extract all elements $v_2(i)$ where $v_1(i) \neq 0$ where $i = 1...\|v_1\|$ and store in v.
VSWAPVV(v, v_1)	swap vectors v and v_1

3.5 Indexed vector operations

VGATHERVX$(v, VX(a, v_1))$	$v(i) = a(v_1(i)),\ i = 1, ..., \|v_1\|$
VXSCATTERV$(VX(a, iv), v_1))$	$a(iv(i)) = v_1(i),\ i = 1, ..., \|v_1\|$

3.6 Elementwise matrix computations

In the following list, m_i for $i = 0, 1, 2, ...$ stands for matrix objects XXX... in some format XXX.

MMAPMM$(m, \oplus, m_1, m_2)$	elementwise appl. of binary operator $\oplus$, results stored in m
MMAPM$(m, \ominus, m_1)$	elementwise application of unary operator $\ominus$, results stored in m
MMAPMV$(m, \oplus, m_1, v_1, d_2)$	map $\oplus$ across dim. d_2 of m_1, results stored in m
MMAPMS$(m, \oplus, m_1, r)$	elementwise apply $\oplus$ to r and all element of m_1, results stored in m
MMAPVV$(m, \oplus, v_1, d_1, v_2, d_2)$	map $\oplus$ across $v_1 \times v_2$, spanning dim.'s d_1, d_2 of m, results stored in m
MCOPYM(m, m_1)	matrix copy of m_1 to m, m and m_1 have the same format
MCNVTM(m, m_1)	like MCOPYM, but formats m and m_1 differ
MEXPANDV(m, v, d)	blow up vector v to a matrix m along dimension d
MTRANSPM(m, m_1)	matrix transpose, result stored in m
MINITS(m, r)	initialize all elements of m by scalar expression r
MASGNS(m, r, i, j)	initialize all elements of m by expression r indexed by a formal row index i and/or column index j

Note that outer product (MOUTERVV) is a special case of MMAPVV.

3.7 Searching and sorting on a matrix

In the following list, rv_1 denotes a matrix row RXXX{..} in some format XXX.

MCOLLM(m, m_1, f, i, j)	filter out all elements $m_1(i, j)$ fulfilling a boolean condition $f(m_1, i, j)$, parameterized by formal row index i and/or formal column index j, results stored in m
MGETSUBM$(m, m_1, s_1, t_1, s_2, t_2)$	extract rectangular submatrix of m_1 in range $(s_1 : t_1, s_2 : t_2)$, results stored in m
MSETSUBM$(m_1, s_1, t_1, s_2, t_2, m_2)$	replace submatrix $m_1(s_1 : t_1, s_2 : t_2)$ by m_2
SGETELM(r, m, i, j)	extract element $m(i, j)$ if it exists, and 0 otherwise, results stored in r
MSETELMS(m, i, j, r)	set element $m(i, j)$ to r
VGETROWM(v, m, i)	extract row i from matrix m, store in v
MSETROWMV(m, i, rv)	set row i in matrix m to rv
VGETCOLM(v, m, i)	extract column i from matrix m, store in v
MSETCOLMV(m, i, cv)	set column i in matrix m to cv
VGETDIAM(v, m, i)	extract diagonal i from matrix m and store in v
MSETDIAMV(m, i, v)	set diagonal i in matrix m to v
MGETLM(m, m_1)	extract left lower triangular matrix (including diagonal) from m_1, store in m
MGETUM(m, m_1)	extract right upper triangular matrix (including diagonal) from m_1, store in m
MPRMROWM(m, v)	permute rows of matrix m with permutation vector v
MPRMCOLM(m, v)	permute columns of m with permutation vector v

3.8 Matrix-vector and matrix-matrix product, decompositions

VMATVECMV(v, r, m, v_1, v_2)	matrix-vector product $v = m \cdot v_1$ where v is initialized to v_2.
VVECMATMV(v, m_1, v_2)	vector-matrix product $v = m_1^T \cdot v_2$.
MMATMULMM(m, m_1, m_2)	matrix-matrix-product $m = m_1 \cdot m_2$
VUSOLVEMV(v, m_1, v_2)	backward subst. $v = m_1^{-1} \cdot v_2,\ m_1$ upper triangular
VLSOLVEMV(v, m_1, v_2)	forward subst. $v = m_1^{-1} \cdot v_2,\ m_1$ lower triangular
VROTVVM(v, v_1, m_2)	Givens rotation
MMLUDM(m, m_1, m_2, p, t)	LU decomposition of m_2, pivot strategy p, drop tolerance t, results stored in m and m_1
VUPDROWM$(v, \oplus, m_1, pr, i, c, space, droptol)$	update row i of m_1 in LU decomp. for pivot row pr, start column c, dense result vector of size $space$, results stored in v

In order to express a transposed matrix-matrix product, the MTRANSP concept has to be applied to the operand matrix to be accessed in transposed order [1]. For dense matrices this can be skipped by toggling the leading dimension.

It is interesting to note that a matrix-vector multiplication for a matrix in CSR format

```
VMATVECMV(..., CSR(...), ...)
```

looks exactly like a transposed matrix-vector multiplication for CSC format

```
VVECMATMV(..., CSC(...), ...)
```

and vice versa. Furthermore, for matrix-vector product the order of nonzero elements within the same row resp. column is not important here, thus the concept variants for CSR and CUR resp. CSC and CUC matrices are equivalent. Thus, for each such pair of equivalent concept variants only one common implementation is required for the back–end.

3.9 I/O concepts

READ and WRITE are the concepts for reading and writing a scalar value to a file.

VREAD(v, F)	read a vector v from file F

[1] The reason why we do not define three more concepts for the combinations of transposed operand matrices is that executing a transpose, if not avoidable, is one order of magnitude less costly than a matrix-matrix product, while the execution time of a transpose is in the same order as a transposed matrix-vector product.

VWRITE(v, F)	write a vector v to file F
MREAD(m, F, f)	read m from file F in file storage format f
MWRITE(m, F, f)	write m to file F in file storage format f

There are various file storage formats in use for sparse matrices, e.g. the Harwell–Boeing file format, the array format, or coordinate format [8].

3.10 Exception slots

For some of the concepts listed above there exist additional slots containing actions specified by the programmer to cover cases when possible exceptions occur. For example, the INV concept (scalar reciprocal) offers a "catch" slot to enter a statement that handles the "division by zero" exception. As another example, for LU decomposition (LUD) on a sparse operand matrix an exception slot indicates what should be done if the allocated space is exceeded.

4 Speculative concept recognition

Safe identification of a sparse matrix operation consists of (1) a test for the syntactical properties of this operation, which can be performed by concept recognition at compile time, and (2) a test for the dynamic properties which may partially have to be performed at run time. Regarding (parallel) code generation, this implies that two versions of code for the corresponding program fragment must be generated: one version branching to an optimized sparse matrix library routine if the test is positive, and a conservative version (maybe using the inspector–executor technique, or just sequential) that is executed otherwise.

4.1 Compile–time concept matching

The static part of our concept matching method is based on a bottom–up rewriting approach using a deterministic finite bottom–up [2] tree–automaton that works on the program's intermediate representation (IR) as an abstract syntax tree or control flow graph, augmented by concept instances and data–flow edges computed during the recognition. Normalizing transformations, such as loop distribution or rerolling of unrolled loops, are done whenever applicable.

The matching rules for the concept idioms to be recognized, called *templates*, are specified as far as possible in terms of subconcept occurrences (see Fig. 2), following the natural hierarchical composition of computations in the given programming language, by applying loops and sequencing to subcomputations. Since at most one template may match an IR node, identification of concept occurrences is deterministic. For efficiency reasons the applicable templates are selected by a hashtable lookup: each rule to match an occurrence of a concept c is indexed by the most characteristic subconcept c' (called the *trigger concept*) that occurs in a matching rule. The graph induced by these edges (c', c) is called the *trigger graph*. Hence, concept recognition becomes a path finding problem in the trigger graph. Matched IR nodes are annotated with concept instances. If working on an abstract syntax tree, a concept instance holds all information that would be required to reconstruct an equivalent of the subtree it annotates.

[2] To be precise, for the unification of objects *within* a matching rule we apply a top–down traversal of (nested) concept instances for already matched nodes.

4.1.1 Vertical matching

Vertical matching proceeds along the hierarchical nesting structure (statements, expressions) of the program's IR, starting with the leaf nodes. Matching a node is only possible when all its children have been matched. The trigger concept used When applying vertical matching to an IR node, the concept that has been matched for its first child is used as the trigger concept.
As a running example, consider the following code excerpt:

```
S1: DO i = 1, n
S2:    b(i) = 0.0
S3:    DO j = first(i), first(i+1)-1
S4:       b(i) = b(i) + a(j) * x(col(j))
       ENDDO
    ENDDO
```

The program's IR (e.g. syntax tree) is traversed bottom–up from the left to the right. Statement S2 is recognized as a scalar initialization, summarized as SINIT(b(i),0.0). Statement S4 is matched as a scalar update computation, summarized as INCR(b(i), MUL(a(j),x(col(j)))). Now the loop around S4 is considered. The index expressions of a and col are bound by the loop variable j which ranges from some loop–invariant value first(i) to some loop–invariant value first(i+1)-1. Thus the accesses to arrays a and col during the j loop can be summarized as vectors V(a,first(i),first(i+1)-1,1) and IV(col,first(i),first(i+1)-1,1). By a template similar to the first one in Fig. 2, the entire j loop is matched as an occurrence of SDOTVVX (dot product with one indexed operand vector); the unparsed program is now

```
S1 : DO i = 1, n
S2': SINIT( b(i), 0.0 );
S3': SDOTVVX( b(i), V(a,first(i),first(i+1)-1,1),
          VX(x,IV(col,first(i),first(i+1)-1,1)),
          b(i));
     ENDDO
```

Although all statements in the body of the i loop are matched, there is no direct way to match the i loop at this point. We must first address the dataflow relations between S2' and S3':

4.1.2 Horizontal matching

Horizontal matching tries to merge several matched IR nodes v_1, v_2, ... belonging to the body of the same parent node (e.g., a loop body). If there is a common concept that covers the functionality of, say, v_i and v_j, there is generally some data flow relation between v_i and v_j that can be used

to guide the matching process. For each summary node we consider the slot entries to be read or written, and compute data flow edges (also called *cross–edges*) connecting slots referring to the same value, e.g., Def–Use chains ("FLOW" cross edges).

Continuing the example above, we obtain that the same value of `b(i)` is written (generated) by the `SINIT` computation in `S2'` and consumed (used and killed) by the `INCR` computation in `S3'`. Note that it suffices to consider the current loop level: regarding horizontal matching, the values of outer loop variables can be considered as constant. Horizontal matching, following the corresponding template (similar to the second template in Fig. 2), "merges" [3] the two nodes and generates a "shared" concept instance:

```
    DO i = 1, n
S'':  SDOTVVX( b(i), V(a,first(i),first(i+1)-
1,1),
       VX(x,IV(col,first(i),first(i+1)-
1,1)), 0.0)
    ENDDO
```

4.2 Speculative concept matching

In order to continue with this example, we now would like to apply vertical matching to the `i` loop. The accesses to `a` and `col` are supposed to be CSR matrix accesses because the range of the loop variable `j` binding their index expressions is controlled by expressions bound by the `i` loop. Unfortunately, the values of the `first` elements are statically unknown. Thus it is impossible to definitively conclude that this is an occurrence of a CSR matrix vector product.

Nevertheless we continue, with assumptions based on syntactic observations only, concept matching in a *speculative* way. We obtain (see also Fig. 3)

```
   <assume first(1)=1>
   <assume monotonicity of V(first,1,n+1,1)>
   <assume injectivity of V(col,first(i),
                 first(i+1)-
1,1) forall i in 1:n>
S: VMATVECMV( V(b,1,n,1),
              CSR(a, IV(first,1,n+1,1),
                  IV(col,first(1),first(n+1)-
1,1),
                  n, first(n+1)-1),
              V(x,1,n,1), VCON(0.0,n) );
```

where the first three lines summarize the assumptions guiding our speculative concept recognition. If they cannot be statically eliminated, these three preconditions would, at code generation, result in three run-time tests being scheduled before or concurrent to the speculative parallel execution of `S` as a CSR matrix vector product. The range of the values in `col` needs not be bound–checked at run time since we can safely assume that the original program runs correctly in sequential.

Now we have a closer look at these pre- and postconditions:

[3] Technically, one node is hidden from further matching and code generation by annotating it with an instance of `EMPTY`, see Fig. 2.

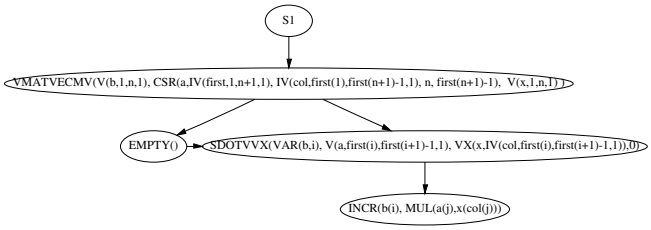

Figure 3: The program graph (abstract syntax tree) of the CSR matrix-vector multiplication code after concept recognition, generated by DOT. As a side-effect of horizontal matching a pseudoconcept "EMPTY" is generated to hide a node from code generation but allow reconstruction of children concepts if desired.

We call an integer vector `iv` *monotonic* over an index range $[L:U]$ at a program point q iff for any control flow path through q, `iv(`i`)`$\leq$`iv(`$i+1$`)` holds at entry to q for all $i \in [L:U-1]$.

We call an integer vector `iv` *injective* over an index range `L:U` at a program point q iff for any control flow path through q, for all $i, j \in$`L:U` holds $i \neq j \implies$ `iv(`i`)` $\neq$ `iv(`j`)` at entry to q. Injectivity of a vector is usually not statically known, but is an important condition that we need to check at various occasions.

We must verify the speculative transformation and parallelization of a recognized computation on a set of program objects which are strongly suspected to implement a sparse matrix A. This consists typically of a check for injectivity of an index vector, plus maybe some other checks on the organizational variables. For instance, for non–transposed and transposed sparse matrix–vector multiplication in CSR or CUR row–compressed format, we have to check that

(1) `first(1)` equals 1,
(2) vector `IV(first,1,`n`+1,1)` is monotonic, and
(3) vectors `IV(col,first(`i`),first(`$i+1$`)-1,1)` are injective for all $i \in \{1, ..., n\}$.

These properties may be checked for separately.

4.3 Speculative loop distribution

Loop distribution is an important normalization applied in the concept recognizer. As an example, consider the following code fragment taken from the `SPARSE-BLAS` [20] routine `DGTHRZ`:

```
   DO 10 i = 1, nz
       x(i)        = y(indx(i))
       y(indx(i)) = 0.0D0
10 CONTINUE
```

In order to definitely recognize (and also in order to parallelize) this fragment, we need to know the values of the elements of array `indx`. Unfortunately, this information is generally not statically available. But similar as for the speculative recognition of sparse matrix operations we speculatively assume that `indx` is injective in the range

1:nz. As now there remain no loop–carried dependencies, we can apply loop distribution [46] to the i loop:

```
<assume injectivity of INDX(1:NZ)>
DO i = 1, nz
    x(i) = y(indx(i))
ENDDO
DO i = 1, nz
    y(indx(i)) = 0.0D0
ENDDO
```

Applying concept matching to each loop separately makes the speculatively matched copy of the program segment look as follows:

```
<assumes injectivity of indx(1:nz)>
VGATHERVX(V(x,1,nz,1), VX(Y,IV(indx,1,nz,1)))
VXSCATTER(VX(y,indx(1,nz,1)), VCON(0.0,n))
```

Speculative loop distribution saves the original program structure for the code generation phase. This allows to generate also the conservative code variant.

4.4 Preservation and propagation of format properties

Even if at some program point we are statically in doubt about whether a set of program objects really implements a sparse matrix in a certain storage format, we may derive static information about some format properties of a speculatively recognized concept instance.

For any concept (or combination of a concept and specific parameter formats) the format property preconditions for its parameter matrices are generally known. If an instance I of a concept c generates a new (sparse) result matrix m, it may also be generally known whether m will have some format properties after execution of I (i.e., a postcondition). Such a property π of m may either hold in any case after execution of an instance of c, i.e. $\pi(m)$ is installed by c. Or, π may depend on some of the actual format properties $\pi_1, \pi_2, ...$ of the operand matrices $m_1, m_2, ...$ In this case, $\pi(m)$ will hold after execution of I only if $\pi_1(m_1)$, $\pi_2(m_2)$ etc. were valid before execution of I. In other words, this describes a propagation of properties $\pi_1(m_1) \wedge \pi_2(m_2) \wedge ... \rightarrow \pi(m)$. Also, it is generally known which properties of operand matrices may be (possibly) deleted by executing an instance of a concept c.

The assumptions, preservations, propagations and deletions of format properties associated with each concept instance are summarized by the program comprehension engine in the form of pre- and postcondition annotations to the concept instances. Note that the preservations are the complementary set of the deletions; thus we renounce on listing them. If existing program objects may be overwritten, their old properties are clearly deleted. Note that the `install` and `propagate` annotations are postconditions that refer to the newly created values. The shorthand `all` stands for all properties considered.

For example, if a certain piece of program has been speculatively recognized as an occurrence of a CSC to CSR conversion concept, annotations are (conceptually) inserted before the concept instance as follows:

```
<assume FirstB(1)=1>
<assume monotonicity of IV(FirstB,1,M,1)>
<assume injectivity of IV(RowB,FirstB(i),FirstB(i+1),1)
                       forall i in 1:M>
<delete all of FirstA>
<delete all of ColA>
<install FirstA(1)=1>
<propagate (monotonicity of IV(FirstB,1,M,1))
 then (monotonicity of IV(FirstA,1,N,1))>
<propagate (monotonicity of IV(FirstB,1,M,1)
 and (injectivity of IV(RowB,FirstB(i),FirstB(i+1),1)
                     forall i in 1:M)
 then(injectivity of IV(ColA,FirstA(i),FirstA(i+1),1)
                     forall i in 1:N)>
MCNVTM( CSR( V(A,1,NZ,1), IV(FirstA,1,N+1,1),
             IV(ColA,1,N,1), N, NZ),
        CSC( V(B,1,NZ,1), IV(FirstB,1,M+1,1),
             IV(RowB,1,M,1), M, NZ) )
```

If applied to an interactive program comprehension framework, these run-time tests correspond to prompting the user for answering yes/no questions about the properties.

4.5 Placing run-time tests

Program points are associated with each statement or concept instance, i.e. with the nodes in the control flow graph after concept matching. In an implementation all properties π of interest for all arrays or array sections A of interest may be stored for any program point q in bitvectors

$ASSUME_{\pi,A}(q) = 1$ iff $\pi(A)$ is `assumed` to hold at entry to q

$DELETE_{\pi,A}(q) = 1$, iff $\pi(A)$ may be `deleted` by execution of q

$INSTALL_{\pi,A}(q) = 1$ iff $\pi(A)$ is `installed` by execution of q. — Propagations are represented by sets

$PROPAGATE_{\pi,A}(q)$ containing all properties π_j of arrays A_j that must hold at entry to q in order to infer $\pi(A)$ at exit of q. — Moreover, we denote by

$TEST_{\pi,A}(q)$ whether a run-time test of property π of array section A has been scheduled immediately before q. When starting the placement of run-time tests, all $TEST_{\pi,A}(q)$ are zero.

For the placement of run-time tests we compute an additional property

$HOLD_{\pi,A}(q)$ which tells whether $\pi(A)$ holds at entry of q. We compute it by a standard data flow technique (see e.g. [46]) iterating over the control flow graph $G = (V, E)$ of the program, using the following data flow equation:

$$
\begin{aligned}
HOLD_{\pi,A}(q) = {} & TEST_{\pi,A}(q) \\
& \vee \bigwedge_{(q',q)\in E} (HOLD_{\pi,A}(q') \wedge \neg DELETE_{\pi,A}(q') \\
& \qquad \vee INSTALL_{\pi,A}(q')) \\
& \vee \bigwedge_{(q',q)\in E} \bigwedge_{(\pi',A')\in PROPAGATE_{\pi,A}(q')} HOLD_{\pi',A'}(q')
\end{aligned}
\tag{1}
$$

For the data flow computation of *HOLD*, we initialize all *HOLD* entries by 1. Since the $DELETE$, *INSTALL*, *PROPAGATE* and *TEST* entries are constants for each π, A, and q, the sequence of the values of $HOLD_{\pi,A}(q)$ during the iterative computation is monotonically decreasing and bounded by zero, thus the data flow computation converges.

Clearly, after all necessary run-time tests have been placed, $HOLD_{\pi,A}(q)$ must fulfill

$$HOLD_{\pi,A}(q) \geq ASSUME_{\pi,A}(q) \quad \text{for all } \pi, A, q$$

in order to ensure correctness of the speculative program comprehension. Thus we arrive, as a very general method, at the following simple nondeterministic algorithm for placing run-time tests:

Algorithm: *placing run-time tests*
(1) **for all** π and A of interest **do**
(2) **for all** q **do** $TEST_{\pi,A}(q) = 0$;
(3) **forever do**
(4) initialize $HOLD_{\pi,A}(q') = 1$ for all program points q'
(5) recompute $HOLD_{\pi,A}(q)$ for all program points q according to equation (1)
(6) **if** $HOLD_{\pi,A}(q) \geq ASSUME_{\pi,A}(q)$ for all q **then break**;
(7) set $TEST_{\pi,A}(q') = 1$ for some suitably chosen program point q'
od
od

The goal is to place the run–time tests in step (7) in such a way that the total run-time overhead induced by them in the speculatively parallelized program is minimized. A very simple strategy is to place a test for $\pi(A)$ *immediately after* each statement q' killing property π of A, i.e. where $KILL_{\pi,A}(q') = 1$, which is defined as follows:

$$KILL_{\pi,A}(q') := DELETE_{\pi,A}(q') \wedge \neg INSTALL_{\pi,A}(q')$$

$$\wedge \neg \bigwedge_{(\pi',A') \in PROPAGATE_{\pi,A}(q')} HOLD_{\pi',A'}(q')$$

Of course, this initial setting will typically introduce superfluous tests. For instance, for a sequence of consecutive killings of $\pi(A)$ it is sufficient to schedule a test for $\pi(A)$ only after the last killing program point, provided that $\pi(A)$ may not be assumed to hold at some point within this sequence. Also, a test for $\pi(A)$ after program point q' is superfluous if $\pi(A)$ is not assumed at any point q'' that may be executed after q'. These optimizations can be carried out in a second phase by another data flow framework. [4]

[4] Alternatively, one may as well start with a test of $\pi(A)$ being inserted immediately before each program point q with $ASSUME_{\pi,A}(q) \wedge \neg HOLD_{\pi,A}(q)$, and then eliminating all tests in a sequence of uses of $\pi(A)$ but the first one (provided that no kill of $\pi(A)$ may occur in between), which is just symmetric to the strategy described above.

4.6 A parallel algorithm for the monotonicity tests

Monotonicity of a vector $x(1:n)$ is very easy to check in parallel. Each of the p processors considers a contiguous slice of x of size at most $\lceil n/p \rceil$. The slices are tested locally; if a processor detects non–monotonicity in its local part, it signals a FAIL and aborts the test on all processors. Otherwise, the values at the boundary to the next upper slice of x are checked concurrently. The test passes if no processor detects non–monotonicity for any pair of neighboring elements.

On a distributed memory architecture, the latter, global phase of this algorithm requires each processor (but the first one) to send the first element of its slice of x to the processor owning the next lower slice; thus each processor (but the last one) receives an element and compares it with the element at the upper boundary of its slice.

4.7 A parallel algorithm for the injectivity tests

A parallel algorithm may reduce the overhead of the injectivity test. For a shared memory parallel target machine we apply an algorithm similar to bucket sort[5] to test injectivity for an integer array `a` of n elements, as it is likely that the elements of `a` are within a limited range (say, $1:m$).[6] We hold a shared temporary array `counter` of m counters, one for each possible value, which are (in parallel) initialized by zero. Each processor k, $k = 1, ..., p$ increments[7] the corresponding counters for the elements $a((k-1)n/p + 1 : kn/p)$. If a processor detects a counter value to exceed 1, it posts a FAIL signal, the test returns FALSE. Otherwise, the test accepts. The test requires m additional shared memory cells. Concurrent write access to the same counter (which may sequentialize access to this location on most shared memory systems) occurs only if the test fails. Thus, the run time is $O((m+n)/p)$.

On a distributed memory system, we use an existing algorithm for parallel sorting of an integer array of size n on a processor network that may be appropriately embedded into the present hardware topology. As result, processor i holds the ith slice of the sorted array, of size n/p. Furthermore, each processor $i > 0$ sends the first element of its slice to its predecessor $i-1$ who appends it as $(n/p+1)$st element to its local slice. Each processor now checks its extended slice for duplicate entries. If the extended slices are injective, then so is the original array. The run time is dominated by the parallel sorting algorithm.

[5] A similar test was suggested in [38].

[6] m is to be chosen as a conservative overestimation of the extent of the compressed matrix dimension which is usually not statically known.

[7] This should be done by an atomic fetch&increment operation such as `mpadd(&(counter[a[j]]),1)` on the SB-PRAM, cf. [27].

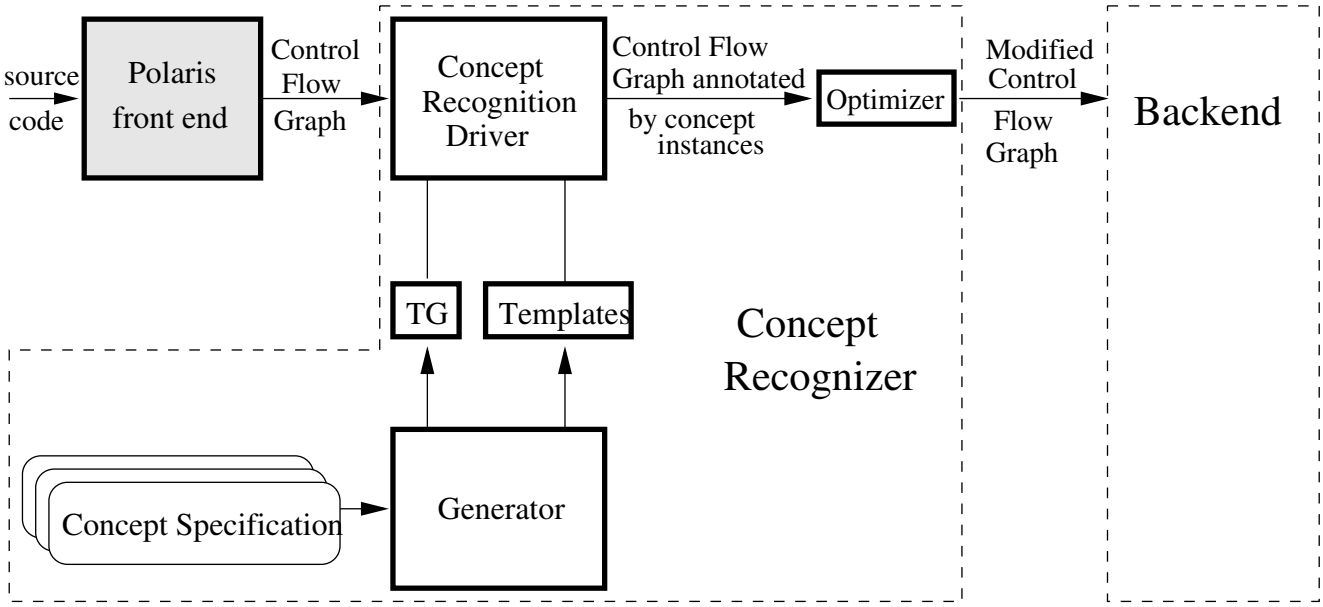

Figure 4: SPARAMAT implementation.

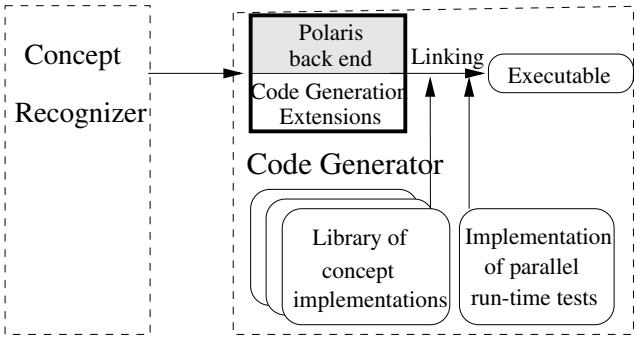

Figure 5: SPARAMAT Code Generator implementation.

5 SPARAMAT Driver Implementation

This section discusses the implementation details of SPARAMAT.

5.1 Overview

SPARAMAT has been developed as using the Polaris Fortran compiler [7, 18]. SPARAMAT is conceptually broken up into two major systems, the concept recognizer and the code generator (see Figure 4 and Figure 5).

Prior to submitting any program for matching, it is necessary to configure the SPARAMAT concept recognition driver by executing the generator program. The generator reads specification files describing concepts and templates and produces C++ files containing the Trigger Graph (TG) (see Section 5.3.1) and template matching functions. The data structure describing the TG and template functions are compiled and linked with the SPARAMAT concept matching core routines to form the driver. The internals of the driver are discussed in Section 5.2.

The analysis of the program begins when the control flow graph of the specified program is passed to the driver from the Polaris front end. The driver, upon completion of analysis, passes to the optimizer the control flow graph annotated with concept instances. The run-time tests, due to pre- and postconditions (see Section 4.2), are optimized in a separate pass whose theory is described in Section 2.2. The optimizer passes the modified control flow graph to the back end that, in turn, uses the attached concept instances and remaining conditions to insert run-time checks and replace matched code with calls to parallel implementations of the concepts.

5.2 Driver

Figure 6 shows the data that is passed between the various subsystems of the driver.

The program control flow graph, G, is first given to the MakeTree system. The matcher of SPARAMAT, like its predecessor PARAMAT, operates on a syntax tree (not a control flow graph). The MakeTree subsystem adds additional pointers to the nodes in the graph to allow tree traversal. DumpTree and DumpConceptTree systems traverse the tree graphs and output DOT files for debugging. DumpConceptTree is similar to DumpTree but concept names are

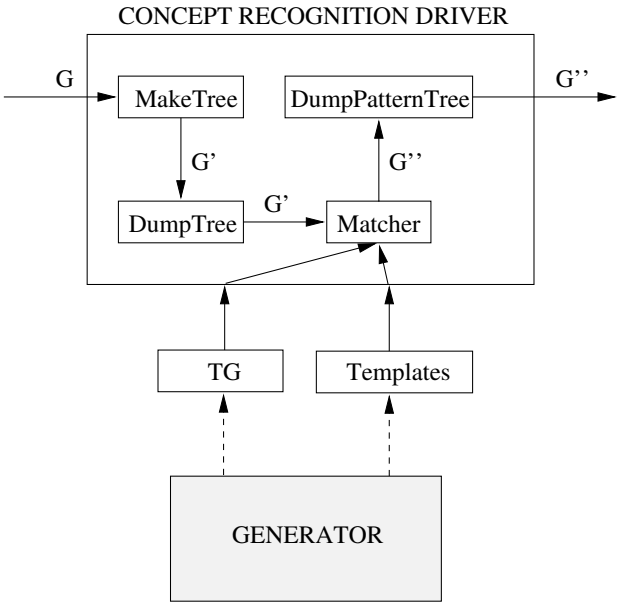

Figure 6: Components of the Driver.

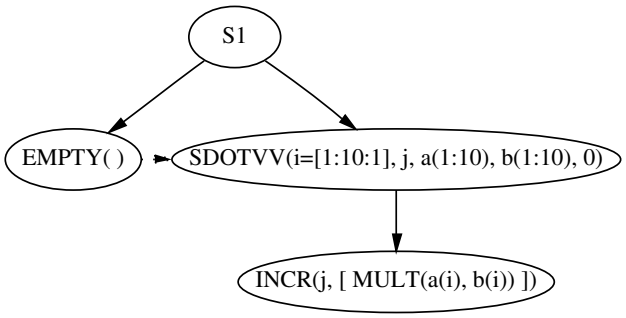

Figure 7: Syntax tree, with patterns for dot product

used as node labels instead of Fortran code.

The Trigger Graph (see Section 5.3.1) and templates previously linked with the matcher and are used for the matching process. The matcher accepts G' and returns G''—the graph G' anointed with recognized concepts.

5.3 Matcher

Matching is the process of analyzing recursively the syntax tree and annotating each node with a concept that summarizes the node. There are two types of matching that take place: Vertical matching and horizontal matching. The implementation details of these matching strategies are discussed here. For more general information see Section 4.1.1 and Section 4.1.2.

Vertical matching takes place during a post-order traversal of the syntax tree. If not all the children of the current node are annotated, then matching for the current node immediately ends. After a node is matched and annotated with a concept instance, cross-edges are attempted at the node. It is not necessary to attempt horizontal matching until a node is annotated by vertical matching.

Horizontal matching is currently implemented only for data-flow cross-edges—the most common type of cross-edge [23]. Because of limitations in Polaris a search backward from the current node to find the source of the data flow has been implemented. The requirements on the source of the data flow and code that is passed over are expressed in specific sets to the search. Variables in sets *inSet* and *outSet* define which variables are expected to be read and written to in the beginning of the data-flow cross-edge. Variables in the sets *notInSet* and *notOutSet* specify those variables that must not be read or written too in the code between the source and the end of the flow cross-edge. See Section 6 for an example of these sets in use.

When horizontal matching succeeds the relevant information is merged with the concept at the node where the search started, and the corresponding other node's concept is saved and replaced with the `EMPTY` concept so the code generator will ignore it and vertical matching can continue at the shared parent node.

5.3.1 Trigger Concepts

For horizontal and vertical matching applying all the template functions to a particular node is impractical and unnecessary. Consider Figure 7: When attempting to match the code located at the `DO` loop the child concept `INCR` permits only certain template functions to match. To reduce the number of template functions applied to a specific node during the matching process, a discriminator, the concept name of the left most child, is used. This discriminator is referred to as the trigger concept.

The same pruning strategy is used for horizontal matching, however the current node is consulted for the trigger concept.

Trigger Graph

The graph consisting of all concepts and edges connecting a concept with its trigger concept for each matching rule, is called the Trigger Graph.

The generator builds the TG from the concept specifications and outputs code that describes the edges and nodes of the graph. Figure 8 shows the TG subgraph for `SDOTVV`. The "leaf" concepts `IVAR` and `RVAR` are concepts for integer and real variables that never have child nodes in the syntax tree, therefore an artifical trigger concept `NONE` is used as the trigger concept for leaf concepts. Each edge is labeled with name of the corresponding template function. A dotted edge denotes a horizontal template.

Note that the TG does not represent all the concepts that

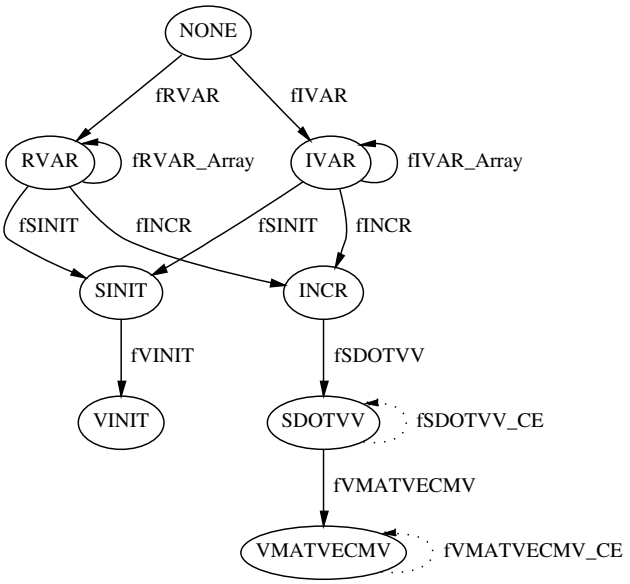

Figure 8: Trigger Graph as far as used for matching the SDOTVV code.

are required for a concept to match. Rather, it shows the concepts that have a specific relationship with an already matched node.

Internally the TG is represented by a table indexed by the trigger concept that yields a list of reachable concepts and thus promising template functions.

5.4 Delayed Format Resolution

In some situations the matrix format might not be clear until a discriminating piece of code is encountered. Until then, it may be necessary to store a *set of* possible formats in the concept instance and postpone the final identification of the format to a later point in the matching process. To support this, we use a special nested concept summarizing the same matrix object in a set of different possible formats within the same slot of the concept instance.

5.5 Descriptors

When computing cross-edges it is necessary to know if two variables intersect. Only when the variable at the definition point intersects with the variable at the use point, does a FLOW cross-edge exist. Determining if scalar variables intersect is straightforward—compare the identifiers (remember there are no pointers in FORTRAN, and all functions are inlined). However, determining intersection for array accesses is not as simple. For two array accesses to intersect they must not only have matching identifiers, but they must intersect in at least one dimension. Further complicating the task is that indices are not limited to constants; they can be variables, arithmetic expressions involving variables or even array accesses.

A descriptor is a data structure that describes the extent of a variable. The extent of a variable is the range of memory locations that are used in the accesses. For example, an array may have ten elements but only the elements between two and five are accessed. A descriptor captures this information for computing the intersection with another variable, also represented by a descriptor. The variable could be a scalar or an n-dimensional array—for the purpose of computing intersection all variables are normalized to a descriptor. A descriptor contains two important pieces of data: The identifier of the variable and an array of range objects that describe the access limits for each dimension. If the variable is a scalar, then only the name field inside the descriptor is set. A range object contains four pieces of data: the lower bound of the access, the upper bound of the access, the stride, and finally a pointer to another descriptor object. If the accessed range is a constant or a non-varying variable, say `c`, then the lower bound and upper bound of the range object is set to `c` and the stride to 0. Otherwise the lower bound and upper bound are set to the expressions substituting the known maximum and minimum for that accessed range. The stride is set also. If the access is an indirect array access, then the descriptor pointer is set to the computed descriptor for that array, and the ranges are set to conservative estimates.

There are two types of intersections that are computed using the descriptors. For the nodes that the cross-edge spans (fill nodes, see Section 6.2), an overestimate of the intersection is necessary to insure that the cross-edge is not broken. However, a conservative intersection must be computed on the cross-edge source node to insure that accesses of the DEF-USE chain match. See [12] and [22] for algorithms for computing the intersection.

6 CSL Specification

The necessity of a concept specification language for SPARAMAT is obvious—generating patterns by hand is a tedious, time consuming and error prone practice best left to automation. A generator would solve these problems allowing the rapid concise specification of concepts and templates in a high-level language. The structure of this language, called CSL (Concept Specification Language), is discussed herein.

6.1 Requirements

To adequately specify concepts and templates certain properties are required.

6.1.1 Concept Templates

A concept is matched by templates: vertical matching templates and, depending on the concept, horizontal (cross-edge) matching templates.

Experience gained from implementing non-trivial sparse matrix concepts by hand, reveals there are five major components to a concept's vertical template function:

- **Matching Conditions**: The conditions that must be met for the concept to match code at the current node. These conditions also consider child nodes' concept instances.
- **Where Conditions**: The expressions on unmatched nodes and the slots of concept instances need to be tested to insure certain properties.
- **Run-time Preconditions**: Run-time conditions that must be met for execution of speculatively generated replacement code, see Section 4.2. This component is not required for concepts that do not have a corresponding library function or that require no speculative matching.
- **Instance Creation**: The code necessary to fill the slots of the concept.
- **Run-time Postconditions**: The run-time conditions that continue to be valid after code execution, assuming that the preconditions held.

The templates for cross-edges have the following major components:

- **Matching Conditions**: Because the driver applies a subset of the cross-edge template functions the conditions verify the starting point of the cross-edge. Additional constraints are necessary on the code between the current node and the start of the data flow edge. For ease of parsing the conditions has been broken up into two sections in the language.
- **Run-time Preconditions**: Run-time preconditions must be met for this horizontal template to match. In general, the preconditions of the cross-edge destination node is unioned with the preconditions of the source node of the cross-edge.
- **Instance Creation**: The template once matched will require new instances for the source and the target node of the cross-edge. The old instances are saved.
- **Run-time Postconditions**: Postconditions if the concept matches.

6.1.2 Debugging

Debugging of a template is inevitable. All hand made changes to the generated code will be lost upon the next invocation of the generator, therefore there must be directives in the language to instruct the generator to insert debugging information in the generated code. The debugging directive can be applied globally to all templates or constrained to particular concepts, or templates.

6.2 CSL Example

The specification of a concept is fairly straightforward. Each component, outlined in Section 6.1.1, translates directly to a construct of the language. Like most other languages, CSL is whitespace insensitive and block structured.

New concepts are created by the keyword `concept` followed by the concept name and a block:

```
concept VMATVECMV
{
  .
  .
  .
}
```

As a running example we use matrix-vector multiplication for a MSR matrix in this section. In order to completely understand the examples that follow it is useful to refer back to the example code for matrix vector multiplication in MSR format in Section 2 on page 4.

The first section inside the concept block is a description of the concept's slots, their usage, and concept group or type.

```
concept VMATVECMV
{
  param (out)  $p_b: vector;
  param (none) $p_rr: range;
  param (in)   $p_mf: matrix;
  param (in)   $p_v: vector;
  param (in)   $p_init: vector;
  .
  .
  .
```

For dependency analysis slots are assigned usage properties: read (`in`), write (`out`), read and write (`inout`) and ignore (`none`). Following the parameter usage is the parameter variable. The '$' character identifies the variable (e.g. `$p_b` in the first slot) as a slot identifier. After the semicolon is the slot type which is either a base type or a concept group. The valid base types of a slot are:

- **range**: A range concept instance holding a loop variable and bounds, for book-keeping purposes.
- **vector**: A `V` concept instance.
- **ivector**: An `IV` instance.
- **xvector**: A `VX` concept instance of an indexed vector access.
- **matrix**: A concept instance for a matrix object.
- **real**: A `CON` or `VAR` instance.
- **int**: An `ICON` or `IVAR` instance.
- **operator**: The concept name of a matched operator (e.g. `MUL`).

If a concept has only one `out` slot, then that slot type is considered the result type of the concept. If the single `out` slot type is one of the *base* types listed above then it is added to a set of concepts with that same return type. Other concepts may define slots using this concept group as the type (the concept group name is simply the base type appended with "_gp"). Hence, in the `pattern` construct (see below), usage of concept instances can be type checked—only concepts in the proper concept group can appear in the correspondingly typed slot.

A sequence of any number of vertical and horizontal templates for this concept are specified within the concept block.

The start of the vertical template block is identified by the `verticalTemplate` keyword. Expressing the matching criteria is broken up into two parts: The `pattern` keyword identifies the section describing the constraints on the children's concept instances and the code on the current node, the `where` keyword identifies the section describing the constraints on slots of the child concept instances and the expressions of the current node:

```
concept VMATVECMV
{
  // b = Av
  param (out)  $p_b: vector;
  param (none) $p_rr: range;
  param (in)   $p_mf: matrix;
  param (in)   $p_v: vector;
  param (in)   $p_init: vector;

  // MSR format, cross-edge needed
  templateVertical
  {
    pattern
       node  DO_STMT $lv=$lb:$ub:$st
      child    SDOTVVX(VAR($s),
                       RANGE($rlv,$rlb,$rub,$rst)
                       V($a, $lb, $ub, $st),
                       VX($v,
                        IV($arr1, $lb1, $ub1, $st1)),
                       VAR($ii))

    where
    .
    .
    .
```

The pattern described is a `DO` loop with one child node annotated by a `SDOTVVX` instance. In the code above, the `$lv` variable binds to the loop variable and `$lb:$ub:$st` variables binds the lower bound, upper bound and the stride expressions. These bindings are used to test properties of code or specific slots in the `where` section. The `where` section is only evaluated if the `pattern` criteria match. The concept instances of the child node can contain nested pattern instances. This powerful feature makes it easy to specify what concept is expected to occur inside and assign names to the slots of that inner concept instance.

The next section of the vertical template is the `where` section. This section tests the run-time static properties not expressed by the `pattern` clause. The C++ code in the `where` section is a boolean expression consisting of calls to library functions to test specific properties of expressions and slots:

```
where
{
  $s == $ii    // uninitialized
  && IsLowerFIRArray($rlb, $lv)
  && IsUpperFIRArray($rub, $lv)
  && IsArrayIndexedOnlyBy($s->GetExpr(), $lv)
  && IsArrayIndexedOnlyBy($a->GetExpr(), $rlv)
  && IsArray(ArrayIndex(0, $v->Expr()))
  && ArrayIndex(0, $v->GetExpr())->array()
     == $rlb->array()
  && IsArrayIndexedOnlyBy(ArrayIndex(0, $v->GetExpr()),
                          $rlv)
}
```

A requirement for the `VMATVECMV` concept for the MSR format is that the `b` vector, the result, is initialized to the product of the diagonal elements of the matrix and the `v` vector elements (see Section 2). The first clause of the expression insures that the initialization slot, `$ii`, is set to the value it has in the preceding code. A cross-edge will be necessary to determine if `b` is set appropriately for this format. A common pattern in code manipulating sparse matrices in a format that stores the indices of the rows in a separate array (see CSR, MSR, CSC and JAD format descriptions) is the pair of expressions `fir(i)` and `fir(i+1)-1` (where `fir` is the array containing the starting indices of each row, and `i` is a loop variable ranging over the number of rows). The `IsLowerFIRArray` and `IsUpperFIRArray` test if the passed expression matches these patterns. The `IsArrayIndexedOnlyBy` function returns true if the first argument is an array where only one index expression is the variable in the second argument. The `IsArray` function merely returns true if the passed expression is an array. Finally the `ArrayIndex` returns an index specified by the first argument of the array named in the second argument. This collection of clauses tries to prove unequivocally that the `DO` loop is linked to the `SDOTVVX` instance by specific variables and the affect of the `DO` loop surrounding the `SDOTVVX` can only mean that these two are part of a `VMATVECMV` concept occurrence. However this is inconclusive, it still needed to show that the result vector is properly initialized. This will be done by a horizontal template that will be specified later.

The next section started by the key word `pre` specifies the run-time conditions that must be met to be certain that the code does indeed implement a matrix vector multiplication in the MSR format. Some of these conditions are possibly removed by the optimizer (see Figure 4); the remaining ones are either inserted by the back end into the generated source code or, if an interactive back end was installed instead, cause the user to be prompted. See Section 4.2 for details.

```
pre
{
  ForAll($lv, $lb, $ub,
         Injectivity($rlb->array(),
                     $rlb, $r->GetEnd()));
  Monotonicity($rlb, $lb, $ub);
}
```

The next section creates an instance of the concept and sets the slots. Appropriately this section is started with the keyword `instance`:

```
instance VMATVECMV(newVector($s,$lv,$lb,$ub,$st),
                   newRange($lv,$lb,$ub,$st),
                   newMSR(newVector($a,1,$ub,1),
                     newIV($arr1,1,$ub+1,1),$ub,$nz),
                   newVector($v,$lv,1,$ub,1),
                   newVMAPV(NEG,
                     newVMAPVV(
                       newVector($s,$lv,$lb,$ub,$st),
                       MUL, newVector($a,1,$ub,1),
                       newVector($arr1,1,$ub,1))))
```

`newRange`, `newVector`, `newV`, `newIV` and `newMSR` are predefined functions that create C++ classes that represent different slot types. The initialization slot, the last slot, is set to be the negative of the pairwise product of the matrix diagonal and the operand vector so that the concept can still be matched as a vector matrix multiply except without the diagonal.

The `post` section that follows is very similar to the `pre` section, except that it defines the run-time conditions that are true after the code implementing the concept has been evaluated. In our example the `post` conditions are the same as the `pre` conditions:

```
pre
{
  ForAll($lv, $lb, $ub,
         Injectivity($rlb->array(),
                     $rlb, $rub));
  Monotonicity($rlb, $lb, $ub);
}
```

The concept, as currently defined, is complete. However, for thoroughness code initializing the result vector must be sought. Referring back to the code example of `VMATVECMV` in Section 2 it is clear that horizontal matching is necessary to locate the initialization code.

```
templateHorizontal
{
  pattern
    sibling ($s) VMAPVV(V($rs, $l1, $u1, $s1), MUL,
                        V($t, $l2, $u2, $s2),
                        V($u, $l3, $u3, $s3))
    fill    ($f)
    node    ($n) VMATVECMV(V($b,$bl,$bu,$bs),
                           RANGE($rlv,$rlb,$rub,$rst),
                           MSR(V($a,$alb,$aub,$ast),
                               IV($f,$flb,$fub,$fst),
                               $ub,$nz),
                           V($v,$vlb,$vub,$vs),
                           VMAPVV(V($b,$bl,$bu,$bs), NEG,
                             VMAPVV(MUL,
                               V($a,$vlb,$vub,$vs),
                               V($v,$vlb, $vub,$vs)))
  .
  .
  .
```

The first section, shown above, is similar to the `pattern` section of a vertical template. Since the nodes involved in a horizontal template are all siblings, the keywords `sibling` and `node` are needed to differentiate the nodes of the cross-edge.

However it is possible that multiple siblings are necessary in the template, therefore the sibling nodes are uniquely labeled so they can be specifically referred to in other sections. The siblings that don't match the concepts in the patterns must also be tested to be sure they do not interfere with the cross-edge. These nodes are referred to as "fill" and are labeled because there might be more than one fill node.

To better explain the horizontal template it is useful to have example code. The following matched FORTRAN code is based on the unmatched code in Section 2 except for the initialization of the output vector that has been moved outside the `i` loop and into its own loop. There is also some extraneous code between the two top level loops:

```
VMAPVV(V(b,1,n,1), MUL, V(a,1,n,1), V(x,1,n,1))
SCOPY(j,b(n+1))
VMATVECMV(V(b,1,n,1),
          RANGE(i,1,n,1),
          MSR(V(a,1,fircol(n+1)-1,1),
              IV(fircol,1,n+1,1),n,nz),
          V(x,1,n,1)
          VMAPV(NEG, VMAPVV(V(b,1,n,1),
                     MUL, V(a,1,n,1), V(x,1,n,1))))
```

The search starts at the node labeled by `node` that contains an instance of the concept `VMATVECMV` that annotates the `DO i` loop. The sibling node is obviously the `VMAPVV`. The fill is therefore the `SCOPY` node—the only sibling located between the nodes connected by the cross-edge. Still to be tested are the run-time properties of the concepts at the ends of the cross-edge and if the fill node interfers with data flow along the cross-edge.

```
where ($s)
{
  $b == $rs
}
where ($n)
{
  $t == $a
  && $u == $v
}
where ($f)
{
  outSet = $b;
  notInSet = $b;
  notOutSet = $b;
  notOutSet += $t;
  notOutSet += $u;
}
```

There `where` keyword is followed by a label created in the `pattern` section. The labels are used to order the evaluation of the `where` clauses. Here, the `where` clause with the `n` label is evaluated immediately to insure that the cross-edge is being considered for the correct node. Next the `where` clause for the sibling (`s`) is evaluated, but on the previous sibling node in the tree. Whenever the `where` clause fails attempting to match the source of the cross-edge, the node is considered fill and the `where` clause for `$f` evaluated. If the node does interfere with the cross-edge, matching fails. The evaluation of the `where` clauses `f` and `s` continues until a match is found, fill code interferes, or previous siblings in the block are exhausted.

Before and after the `instance` section are normally the `pre` and `post` conditions. The conditions for this cross-edge can be dropped because they are exactly the same as the conditions for the `VMATVECMV` concept instance previously matched for node `$n`.

The last section of the horizontal template, the `instance` section, creates new instances for the source and

destination of the cross-edge. In this example, the concept and slots remain the same except for the initialization slot of the VMATVECMV instance which is set to `VCON(0.0)`. This `VCON(0.0)` signifies that the VMATVECMV concept is complete—the result vector is properly initialized.

```
instance ($f) EMPTY()

instance ($n)
  VMATVECMV($p_b, $p_rr, $p_mf, $p_v, newVCON(0.0,n))
```

7 Related work

Several automatic concept comprehension techniques have been developed over the last years. These approaches vary considerably in their application domain, purpose, method, and status of implementation.

General concept recognition techniques for scientific codes have been contributed by Snyder [44], Pinter and Pinter [35], Paul and Prakash [34], diMartino [30] and Keßler [24]. Some work focuses on recognition of induction variables and reductions [1, 36, 19] and on linear recurrences [11, 39]. General techniques designed mainly for non–numerical codes have been proposed by Wills et al. [40] and Ning et al. [21, 28].

Concept recognition has been applied in some commercial systems, e.g. EAVE [9] for automatic vectorization, or CMAX [42] and a project at Convex [31] for automatic parallelization. Furthermore there are several academic applications [24, 30] and proposals for application [10, 3] of concept recognition for automatic parallelization. Today, most commercial compilers for high–performance computers are able to perform at least simple reduction recognition automatically.

A more detailed survey of these approaches and projects can be found e.g. in [24] or [16].

Our former PARAMAT project (1992–94) [24] kept its focus on dense matrix computations only, because of their static analyzability. The same decision was also made by other researchers [35, 30, 31, 3] and companies [42] investigating general concept recognition with the goal of automatic parallelization. According to our knowledge, there is currently no other framework that is actually able to recognize sparse matrix computations in the sense given in this paper.

8 Conclusion and future work

We have described a framework for applying program comprehension techniques to sparse matrix computations and its implementation. We see that it is possible to perform speculative program comprehension even where static analysis does not provide sufficient information; in these cases the static tests on the syntactic properties (pattern matching) and consistency of the organizational variables are complemented by user prompting or run-time tests whose placement in the code can be optimized by a static data flow framework.

If applied to parallel code generation, speculatively matched program parts may be optimistically replaced by suitable parallel library routine calls, together with the necessary (parallel) run-time tests. Only if the tests are passed, parallel execution may continue with the optimized parallel sparse matrix library routine. Otherwise, it must fall back to a conservative code variant.

Our automatic program comprehension techniques for sparse matrix codes can also be used in a non–parallel environment, e.g. for program flow analysis, for program maintenance, debugging support, and for more freedom of choice for a suitable data structure for sparse matrices.

Current work on the SPARAMAT implementation focuses on CSL and the generator. Once operational, we will implement the complete list of concepts given in Section 3 with the most important templates.

References

[1] Z. Ammarguellat and W. L. Harrison III. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 283–295. ACM Press, June 20–22 1990.

[2] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.

[3] S. Bhansali, J. R. Hagemeister, C. S. Raghavendra, and H. Sivaraman. Parallelizing sequential programs by algorithm-level transformations. In V. Rajlich and A. Cimitile, editors, *Proc. 3rd IEEE Workshop on Program Comprehension*, pages 100–107. IEEE Computer Society Press, Nov. 1994.

[4] A. J. C. Bik. *Compiler support for sparse matrix computations*. PhD thesis, Leiden University, 1996.

[5] A. J. C. Bik and H. A. G. Wijshoff. Automatic Data Structure Selection and Transformation for Sparse Matrix Computations. *IEEE Trans. Parallel and Distrib. Syst.*, 7(2):109–126, Feb. 1996.

[6] R. Bixby, K. Kennedy, and U. Kremer. Automatic Data Layout Using 0-1 Integer Programming. Technical Report CRPC-TR93349-S, Center for Research on Parallel Computation, Rice University, Houston, TX, Nov. 1993.

[7] W. Blume et al. Polaris: The next generation in parallelizing compilers,. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, Aug. 1994.

[8] R. Boisvert, R. Pozo, K. Remington, and J. Dongarra. Matrix-market: a web resource for test matrix collections. In R. B. et al., editor, *The Quality of Numerical Software: Assessment and Enhancement*, pages 125–137. Chapman and Hall, 1997.

[9] P. Bose. Interactive Program Improvement via EAVE: An Expert Adviser for Vectorization. In *Proc. ACM Int. Conf. Supercomputing*, pages 119–130, July 1988.

[10] T. Brandes and M. Sommer. A Knowledge-Based Parallelization Tool in a Programming Environment. In *Proc. 16th Int. Conf. Parallel Processing*, pages 446–448, 1987.

[11] D. Callahan. Recognizing and parallelizing bounded recurrences. In *Proc. 4th Annual Workshop on Languages and Compilers for Parallel Computing*, 1991.

[12] D. Callahan and K. Kennedy. Analysis of Interprocedural Side Effects in a Parallel Programming Environment. In *First Int. Conf. on Supercomputing, Athens (Greece)*. Springer LNCS 297, June 1987.

[13] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman and MIT Press, 1989.

[14] D. Conway. *Parse::RecDescent - Generate Recursive-Descent Parsers*. Department Of Computer Science and Software Engineering, Monash University, 1.61 edition, 1997.

[15] K. M. Decker and R. M. Rehmann, editors. *Programming Environments for Massively Parallel Distributed Systems*. Birkhäuser, Basel (Switzerland), 1994. Proc. IFIP WG 10.3 Working Conf. at Monte Verita, Ascona (Switzerland), Apr. 1994.

[16] B. DiMartino and C. W. Keßler. Program comprehension engines for automatic parallelization: A comparative study. In I. Jelly, I. Gorton, and P. Croll, editors, *Proc. First Int. Workshop on Software Engineering for Parallel and Distributed Systems*, pages 146–157. London: Chapman&Hall, March 25–26 1996.

[17] I. S. Duff. `MA28` – a set of Fortran subroutines for sparse unsymmetric linear equations. Tech. rept. AERE R8730, HMSO, London. Source code available via netlib [33], 1977.

[18] K. A. Faigin, S. A. Weatherford, J. P. Hoeflinger, D. A. Padua, and P. M. Petersen. The Polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, Oct. 1994.

[19] M. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. Program. Lang. Syst.*, 17(1):85–122, Jan. 1995.

[20] R. G. Grimes. `SPARSE-BLAS` basic linear algebra subroutines for sparse matrices, written in Fortran77. Source code available via netlib [33], 1984.

[21] M. Harandi and J. Ning. Knowledge-based program analysis. *IEEE Software*, pages 74–81, Jan. 1990.

[22] P. Havlak and K. Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Trans. Parallel and Distrib. Syst.*, 2(3):350–359, July 1991.

[23] C. W. Keßler. Symbolic Array Data Flow Analysis and Pattern Recognition in Dense Matrix Computations. In *[15]*, pages 57–68, Apr. 1994.

[24] C. W. Keßler. Pattern-driven Automatic Parallelization. *Sci. Progr.*, 5:251–274, 1996.

[25] C. W. Keßler. Applicability of Program Comprehension to Sparse Matrix Computations. In *Proc. 3rd Int. Euro-Par Conf.* Springer LNCS, Aug. 1997.

[26] C. W. Keßler. Applicability of Automatic Program Comprehension to Sparse Matrix Computations. In P. Fritzson, editor, *Proc. 7th Workshop on Compilers for Parallel Computers*, pages 218–230. Technical Report of the University of Linköping, Sweden, June 1998.

[27] C. W. Keßler and H. Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. J. Parallel Programming*, 25(1):17–50, Feb. 1997.

[28] W. Kozaczynski, J. Ning, and T. Sarver. Program concept recognition. In *Proc. 7th Knowledge-Based Software Engineering Conf. (KBSE'92)*, pages 216–225, 1992.

[29] K. S. Kundert. `SPARSE 1.3` package of routines for sparse matrix LU factorization, written in C. Source code available via netlib [33], 1988.

[30] B. D. Martino and G. Iannello. Pap Recognizer: a Tool for Automatic Recognition of Parallelizable Patterns. In *Proc. 4th IEEE Workshop on Program Comprehension*. IEEE Computer Society Press, Mar. 1996.

[31] R. Metzger. Automated Recognition of Parallel Algorithms in Scientific Applications. In *IJCAI-95 Workshop Program Working Notes: "The Next Generation of Plan Recognition Systems"*. sponsored jointly by IJCAII/AAAI/CSCSI, Aug. 1995.

[32] R. Mirchandaney, J. Saltz, R. M. Smith, D. M. Nicol, and K. Crowley. Principles of run-time support for parallel processors. In *Proc. 2nd ACM Int. Conf. Supercomputing*, pages 140–152. ACM Press, July 1988.

[33] NETLIB. Collection of free scientific software. Accessible by anonymous ftp to `netlib2.cs.utk.edu` or `netlib.no` or e-mail `"send index"` to `netlib@netlib.no`.

[34] S. Paul and A. Prakash. A Framework for Source Code Search using Program Patterns. *IEEE Trans. on Software Engineering*, 20(6):463–475, 1994.

[35] S. S. Pinter and R. Y. Pinter. Program Optimization and Parallelization Using Idioms. In *Proc. ACM SIGPLAN Symp. Principles of Programming Languages*, pages 79–92, 1991.

[36] B. Pottenger and R. Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. In *Proc. 9th ACM Int. Conf. Supercomputing*, pages 444–448, July 1995.

[37] W. H. Press, S. A. Teukolski, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C — The Art of Scientific Computing, second edition*. Cambridge University Press, 1992.

[38] L. Rauchwerger and D. Padua. The Privatizing DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. In *Proc. 8th ACM Int. Conf. Supercomputing*, pages 33–43. ACM Press, July 1994.

[39] X. Redon and P. Feautrier. Detection of Recurrences in Sequential Programs with Loops. In *Proc. Conf. Parallel Architectures and Languages Europe*, pages 132–145. Springer LNCS 694, 1993.

[40] C. Rich and L. M. Wills. Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Software*, pages 82–89, Jan. 1990.

[41] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations, Version 2. Research report, University of Minnesota, Minneapolis, MN 55455, June 1994.

[42] G. Sabot and S. Wholey. Cmax: a Fortran Translator for the Connection Machine System. In *Proc. 7th ACM Int. Conf. Supercomputing*, pages 147–156, 1993.

[43] M. K. Seager and A. Greenbaum. Slap: Sparse Linear Algebra Package, Version 2. Source code available via netlib [33], 1989.

[44] L. Snyder. Recognition and Selection of Idioms for Code Optimization. *Acta Informatica*, 17:327–348, 1982.

[45] M. Ujaldon, E. L. Zapata, S. Sharma, and J. Saltz. Parallelization Techniques for Sparse Matrix Applications. *J. Parallel and Distrib. Comput.*, 38(2), Nov. 1996.

[46] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series. Addison-Wesley, 1990.

[47] Z. Zlatev. *Computational Methods for General Sparse Matrices*. Kluwer Academic Publisher, 1991.

[48] Z. Zlatev, J. Wasniewsky, and K. Schaumburg. *Y12M - Solution of Large and Sparse Systems of Linear Algebraic Equations*. Springer LNCS vol. 121, 1981.

A Example code

As an example, consider the following Fortran implementation of a Hopfield neural network simulation based on a sparse matrix describing the synapse interconnections of the neurons.

```
   program main
   integer wfirst(21), wcol(20), i, j, k, n, nz
   real wdata(100), xst(20), stimul(20), val(20),
   real alpha, beta, tinv, mexp, pexp, accum, tanhval

c  read test matrix in csr format:
   wfirst(1)=1
   read(*,*) n
   do i = 1, n
     read(*,*) k
     wfirst(i+1) = wfirst(i)+k
     do j = wfirst(i),wfirst(i+1)-1
       read(*,*) wdata(j)
       read(*,*) wcol(j)
     enddo
   enddo
   nz = wfirst(n+1)-1

c  simulate a hopfield network (learning online)
c  with interconnection matrix (wdata,wcol,wfirst):
   niter=100
   alpha=0.2
   beta=0.7
   tinv=1.0
   do i = 1, n
     stimul(i) = -1.0
     xst(i) = 0.0
   enddo
   do k = 1, niter
     do i = 1, n
       accum = 0.0
       do j = wfirst(i), wfirst(i+1)-1
         accum = accum + wdata(j)*xst(wcol(j))
       enddo
       val(i) = beta*accum + alpha*stimul(i)
     enddo
     do i = 1, n
       pexp = exp(val(i))
       mexp = exp(-val(i))
       tanhval = (pexp-mexp) / (pexp+mexp)
       xst(i) = tanhval
     enddo
     do i = 1, n
       do j = wfirst(i), wfirst(i+1)-1
         wdata(j)=wdata(j)+tinv*(xst(i)*xst(wcol(j)))
       enddo
     enddo
     tinv = tinv * 0.9
   enddo
   do i = 1, n
     write (*,*) xst(i)
   enddo
   end
```

After applying concept matching and optimizing the format property conditions, the unparsed program looks as follows:

```
program main

integer wfirst(21), wcol(20), k, n, nz
real wdata(100), xst(20), stimul(20), val(20), tinv
real mexp(20), pexp(20), accum(20)

MREAD( CSR(V(wdata,1,nz,1),IV(wfirst,1,n+1,1),
           IV(col,1,n,1),n,nz, stdin, _simplehb) )

<assume monotonicity of wfirst(1:n+1)>
<assume injectivity of wcol(wfirst(i):wfirst(i+1)) forall i in 1:n>

SINIT(tinv,1.0)
VINIT( V(stimul,1,n,1), -1.0)
VINIT( V(xst,1,n,1), 0.0)
do k = 1, 100
  VMATVECMV( V(accum,1,n,1),
             CSR( V(wdata,1,nz,1),IV(wfirst,1,n+1,1), IV(col,1,n,1),n,nz),
             V(xst,1,n,1), VCON(0.0,n)
  VMAPVV( V(val,1,n,1), ADD, VMAPVS(MUL, V(accum,1,n,1), 0.7),
                             VMAPVS(MUL, V(stimul,1,n,1), 0.2))
  VMAPV( V(pexp,1,n,1), EXP, V(val,1,n,1))
  VMAPV( V(mexp,1,n,1), EXP, VMAPV(NEG, V(val,1,n,1)))
  VMAPVV( V(xst,1,n,1), DIV,
          VMAPVV(ADD, V(pexp,1,n,1), VMAPV(NEG, V(mexp,1,n,1))),
          VMAPVV(ADD, V(pexp,1,n,1), V(mexp,1,n,1)))
  MOUTERVV( CSR( V(wdata,1,nz,1),IV(wfirst,1,n+1,1), IV(col,1,n,1),n,nz),
            VMAPVS( MUL, V(xst,1,n,1), tinv ), V(xst,1,n,1) )
  SCAL(tinv, 0.9)
enddo
VWRITE( V(xst,1,n,1), stdout )
end
```

B CSL Grammar

The grammar below is based on Damian Conway's excellent RecDescent Perl module [14] that builds recursive descent parsers from grammar. Here is a legend to help understand the grammar (for an excellent treatment of RecDescent see man page `Parse::RecDescent(1)`):

```
rule(?)       = Match one-or-zero times
rule(s)       = Match one-or-more times
rule(s?)      = Match zero-or-more times
extract_bracketed(text, end) = extract a token from the string in text
                between the second argument and its matching character.
```

```
start:           (debug | concept)(s)
concept:         'concept' concept_name concept_type(?)
                 '{' slotdecl(s?) vertical(s?) horizontal(s?) '}'
concept_type:    ':' slotbasetype
slotdecl:        slotkeywd '(' data_direction ')' sid ':' slottype ';'
slotkeywd:       'param' | 'slot' # slot or param both are OK
data_direction:  'in' | 'out' | 'inout' | 'none'
slotbasetype:    'range' | 'vector' | 'matrix' | 'realconst' | 'integer'
                 | 'concept' | 'xvector'| 'intconst' | 'real'
                 | 'symtabentry'
slottype:        concept_gp | slotbasetype
concept_gp:      /[a-zA-Z]+\_gp/
vertical:        'templateVertical'   '{' v_pattern v_where(?)
                 pre_conditions(?) v_instance post_conditions(?) '}'
horizontal:      'templateHorizontal' '{' h_pattern h_where(s)
                 pre_conditions(?) h_instance post_conditions(?) '}'
id:              /[a-zA-Z_]\w*/
v_pattern:       'pattern'  v_node child(s?)
v_where:         'where'    code
pre_conditions:  'pre' code
v_instance:      'instance' concept_name
                 { ::extract_bracketed($text, ')'); }
h_instance:      'instance' nid concept_name
                 { ::extract_bracketed($text, ')'); }
post_conditions: 'post' code
h_pattern:       'pattern'  sibling_and_fill(s?) h_node
                            sibling_and_fill(s?)
sibling_and_fill: sibling fill
h_where:         'where'  nid code
v_node:          'node'  code_pattern
child:           'child' concept_pattern
code:            { ::extract_bracketed($text, '}'); } # remove start/end
cond_stmt:       'ForAll' '(' forall_args ')' ';' |
                 'Monontonicity' '(' monon_args ')' ';'
forall_args:     sid ',' cond_expr ',' cond_expr ','
                 cond_expr
monon_args:      sid ',' cond_expr ',' cond_expr
cond_expr:       cond_term | 'Injectivity' '(' inject_args ')'
cond_term:       sid l1 | sid l1 | cond_array l1 | cond_num l1
l1:              cond_arith_op cond_term | # empty
cond_array:      sid '[' cond_expr ']' | sid '[' cond_expr ']'
cond_num:        /[0-9]+/
inject_args:     cond_expr ',' cond_expr ',' cond_expr
cond_arith_op:   '+' | '-' | '*' | '/'
sibling:         'sibling' nid concept_pattern
fill:            'fill'    nid
h_node:          'node'    nid concept_pattern
nid:             '(' sid ')'       # node id
code_pattern:    stmt_pattern | expr_pattern
concept_pattern: concept_name '(' sid_list(?) ')' | sid
sid:             '$' id            # slot id
stmt_pattern:    do_stmt | assign_stmt | if_stmt
expr_pattern:    number_expr  | var_expr | arith_expr | comma_expr
concept_name:    /[A-Z][A-Z0-9]+/    # action: verify id against concept list
sid_list:        sid_or_concept ',' sid_list | sid_or_concept
sid_or_concept:  sid | concept_pattern
do_stmt:         'DO_STMT' sid '=' sid ':' sid ':' sid
assign_stmt:     'ASSIGN_STMT' concept_pattern '=' concept_pattern
if_stmt:         'IF_STMT' sid
number_expr:     int_number | real_number
var_expr:        scalar_var | array_var
arith_expr:      'ADD_OP' | 'MULT_OP'
comma_expr:      'COMMA_OP'
int_number:      'INTEGER_CONSTANT_OP' sid
real_number:     'REAL_CONSTANT_OP' sid
scalar_var:      'ID_OP' sid
array_var:       'ARRAY_REF_OP'
debug:           '#DEBUG' debug_arg
debug_arg:       'OUTPUT_ON'  |
                 'OUTPUT_OFF'
```