RosePolly:manual

Athanasios Konstantinidis

April 24, 2013

Contents

Chapter 1

Introduction to the Polyhedral Model

The polyhedral model is an alternative to abstract syntax trees (AST) that enables compilers to analyse and transform programs by utilizing well known mathematical abstractions and libraries like systems of affine inequalities, integer linear programming, Fourier-Motzkin elimination etc. The syntax trees that can be substituted by the polyhedral model are called *scops* or *Static Control Programs* and they are essentially computation kernels with statically predictable control-flow. In practice, if we rely on a high-level syntactic representation like the one used in ROSE (i.e. SAGE III) [1] then *scops* are restricted to arbitrary-nested for-loops where each loop must have affine loop bounds on problem sizes and outer loop induction variables as well as conditionals with conditions following the same rule. In addition, memory accesses are also required to have statically predictable footprints in order to take full advantage of the power of the polyhedral model even though this is not strictly necessary (see Section 4.6).

One of the primary motivations for using the polyhedral model is to avoid a classic restriction of ASTs with regard to program transformations namely the phase ordering restriction [11]. In particular, multiple AST transformations are typically performed as ordered sequences of individual transformation steps each one relying on syntactic pattern-matching analysis. The ordering of these steps becomes an important concern if we consider their impact on the size and complexity of the AST. However, if we use a mathematical representation like the polyhedral model, this concern goes away since we can represent/compose arbitrary sequences of transformations as a single affine transformation that can be applied in a single step. Furthermore, with a polyhedral representation we are able to capture all the runtime instances of statements as well as precise memory access information which is a property that admits to powerful and robust analysis engines.

[1] It is worth noting that SSA based compilers like LLVM can lift these syntactic restrictions [12].

The example of Figure 1.1 shows an ADI kernel (Alternating Direction Implicit) that has been analysed and transformed using the polyhedral model. Notice that a combination of loop fusion and loop skewing has been applied to the original kernel simply through the affine partition mappings of Figure 1.1b. These mappings have been derived from a sophisticated polyhedral scheduling algorithm namely Pluto [4, 5] that maximizes parallelism and locality through systems of uniform recurrence equations [6, 10, 9]. The next paragraph will discuss general details about the implementation and compilation flow of polyhedral compilation frameworks.

```
for ( t = 0 ; t < TT ; t++ ) {
  for ( i1 = 0 ; i1 < NN ; i1++ ) {
    for ( i2 = 1 ; i2 < NN ; i2++ ) {
      S0(t,i1,i2) : X[i1][i2] = X[i1][i2] - X[i1][i2-1]*A[i1][i2]/B[i1][i2-1];
      S1(t,i1,i2) : B[i1][i2] = B[i1][i2] - A[i1][i2]*A[i1][i2]/B[i1][i2-1];
    }
  }
  for ( i1 = 1 ; i1 < NN ; i1++ ) {
    for ( i2 = 0 ; i2 < NN ; i2++ ) {
      S2(t,i1,i2) : X[i1][i2] = X[i1][i2] - X[i1-1][i2]*A[i1][i2]/B[i1-1][i2];
      S3(t,i1,i2) : B[i1][i2] = B[i1][i2] - A[i1][i2]*A[i1][i2]/B[i1-1][i2];
    }
  }
}
```

(a) Original program

$$
\begin{aligned}
F_{S_0} &= (t + i1 + i2, t + i1, t + i2, 0) \\
F_{S_1} &= (t + i1 + i2, t + i1, t + i2, 0) \\
F_{S_2} &= (t + i1 + i2, t + i1, t + i2 + 1, 1) \\
F_{S_3} &= (t + i1 + i2, t + i1, t + i2 + 1, 1)
\end{aligned}
$$

(b) Affine transforms

```
for ( c1=1 ; c1<=2*NN+TT-3 ; c1++ ) {
  if ( c1 <= NN-1 ) {
    S0(0,0,c1);
    S1(0,0,c1);
  }
  par for ( c2=max(1,c1-NN+1) ; c2<=min(c1-1,NN+TT-2) ; c2++ ) {
    if ( c2 <= NN-2 ) {
      S0(0,c2,c1-c2);
      S1(0,c2,c1-c2);
    }
    if ( c2 >= NN-1 ) {
      S0(c2-NN+1,NN-1,c1-c2);
      S1(c2-NN+1,NN-1,c1-c2);
    }
    par for ( c3=max(c1-NN+2,c1-c2+1) ; c3<=min(c1,c1-c2+TT-1) ; c3++ ) {
      S0(-c1+c2+c3,c1-c3,c1-c2);
      S1(-c1+c2+c3,c1-c3,c1-c2);
      S2(-c1+c2+c3-1,c1-c3+1,c1-c2);
      S3(-c1+c2+c3-1,c1-c3+1,c1-c2);
    }
    if ( c2 >= TT ) {
      S2(TT-1,c2-TT+1,c1-c2);
      S3(TT-1,c2-TT+1,c1-c2);
    }
  }
  if ( c1 <= NN+TT-2 ) {
    par for ( c3=max(1,c1-NN+2) ; c3<=min(TT,c1) ; c3++ ) {
      S2(c3-1,c1-c3+1,0);
      S3(c3-1,c1-c3+1,0);
    }
  }
}
```

(c) Transformed program

Figure 1.1: This example demonstrates the power of the polyhedral model by showing how a composition of complex transformations like loop-fusion and loop-skewing can be applied using simple affine expressions.

Chapter 2

Background

The polyhedral representation of a *scop* (Static Control Program) consists of a list of computation statements $S_i : i = 0, \ldots, N$ each one carrying three main pieces of information namely the domain D_{S_i}, the schedule F_{S_i} and the access functions Π_{ij} for each memory reference $j = 0, \ldots, M_i$ of S_i. The domain D_{S_i} of each statement S_i denotes an execution domain through a finite set of affine inequalities i.e. a polyhedron. The schedule F_{S_i} denotes an execution order through a multi-dimensional affine transformation from the original domain D_{S_i} to a new transformed one[1] with a new lexicographic ordering. Finding good schedules that could optimize various properties at the same time is the main objective and power of a polyhedral compilation framework. In order to do that we need to make sure that we respect all dependencies between memory accesses. Therefore, in addition to the list of computation statements a polyhedral representation consists of a list of dependencies that would enable us to analyse and optimize a *scop*. Unlike conventional dependence analysis [19, 1, 2] polyhedral dependencies are exact in the sense that they represent directed edges from one run-time instance to another. The next four paragraphs discuss these four concepts in more detail while Figure 2.1 depicts a UML view of the polyhedral model.

2.1 The Domain

Formally, if a statement S is surrounded by m loops then its execution domain is the convex set of all vectors $\mathbf{x_S} \in \mathbb{Z}_m$ that satisfy (2.1) where $\mathbf{x_S}$ is the iteration vector of S, $\mathbf{n}$ a vector of problem size parameters and D_S an integer matrix of coefficients. Clearly, since iteration vectors can only have integral values, inequality (2.1) defines an integer polyhedron or in other words a Z-Polyhedron. The example of Figure 2.2 shows a simple loop nest along with a set of 6 affine inequalities corresponding to the loop bounds and conditionals surrounding S_0.

[1] Polyhedral transformations are not restricted to unimodular ones [16, 3]

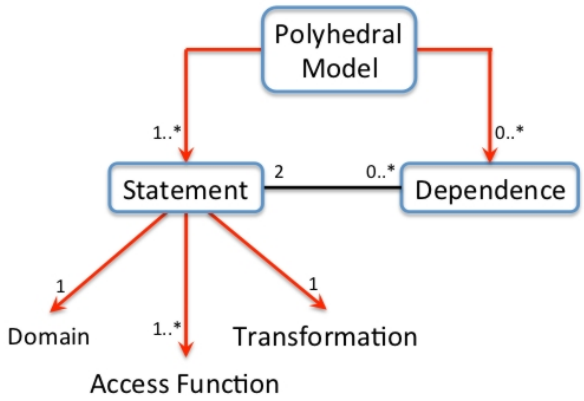


Figure 2.1: Basic UML structure of the polyhedral model

$$D_S \cdot \begin{pmatrix} \mathbf{x_S} \\ \mathbf{n} \\ 1 \end{pmatrix} \geq 0 \tag{2.1}$$

This set of affine inequalities can be used to construct the execution domain D_{S_0} of S_0 shown in Figure 2.2c. Notice that the disjunctive conditional surrounding S_0 results in a concave Z-Polyhedron which is typically implemented as a linked list of 2D integer matrices where each matrix represents a convex Z-Polyhedron according to (2.1) and the entire domain is a disjunction of such polyhedra (linked list) [2].

```
for ( i = 0 ; i < N ; i++ )
  for ( j = 5 ; j < N-5 ; j++ )
    if ( i >= 5 || j <= 10 )
      S0(i,j)
```

(a)

$$\begin{aligned} i &\geq 0 \\ N-1-i &\geq 0 \\ j-5 &\geq 0 \\ N-1-j &\geq 0 \\ i-5 &\geq 0 \\ 10-j &\geq 0 \end{aligned}$$

(b)

$$D_{S_0} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & 1 & 0 & -5 \\ 0 & -1 & 1 & -1 \\ 1 & 0 & 0 & -5 \end{bmatrix} \xrightarrow{\text{OR}} \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & 1 & 0 & -5 \\ 0 & -1 & 1 & -1 \\ 0 & -1 & 0 & 10 \end{bmatrix}$$

(c)

Figure 2.2: This example shows a syntactic form of a loop-nest (a), a finite set of affine inequalities corresponding to that loop-nest (b), and the actual polyhedral representation (c).

[2] Of course these implementation details are not exposed to the user by the RosePolly API

2.2 The Schedule

The default execution order of all the points inside an execution domain is the lexicographic one. For example iteration $(0, 4)$ is executed before $(1, 0)$ according to the original execution domain of Figure 2.3. If we wish to change that order we would need to define an affine transformation that would map each point to a new transformed domain effectively changing the lexicographic order and as a result the execution order. Figure 2.3 shows an example of a loop skewing transformation implemented as an affine transformation while Figure 2.4 shows a loop fusion example.

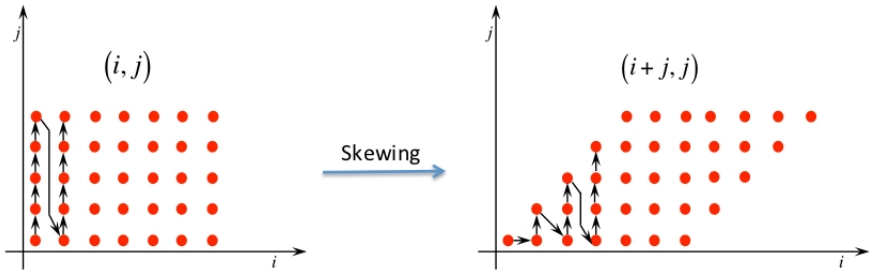


Figure 2.3: Loop skewing in the polyhedral model

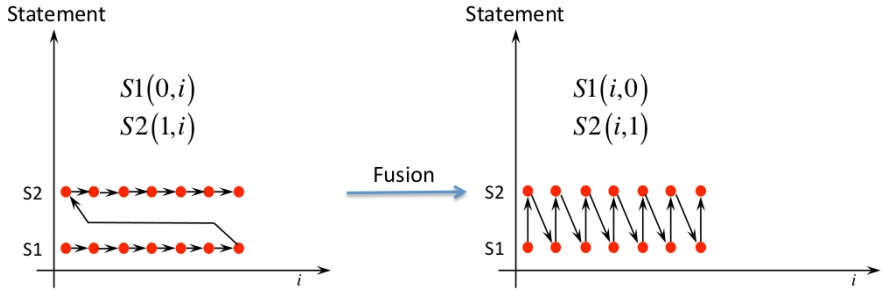


Figure 2.4: Loop fusion in the polyhedral model

2.3 Memory Access Functions

Memory access functions are affine transformations that map each point of an execution domain to a point in the data space of the corresponding variable. Let Π_{ij} be an affine transform representing the memory access function of an array reference $j = 0 \ldots, N_i$ of a statement $S_i : i = 0, \ldots, M$. Figure 2.5 shows an example of memory access functions interpreted as affine transformation

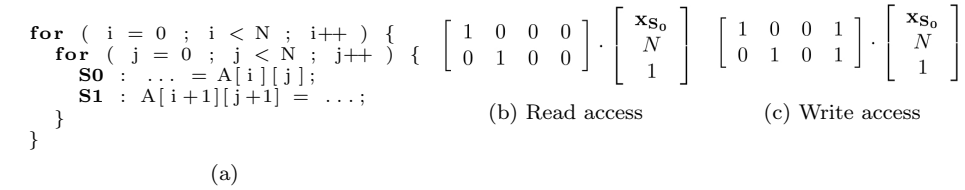


Figure 2.5: Example showing how array access functions are handled by the polyhedral model

matrices. This representation allows us to perform precise dependence analysis (see Section 2.4), temporal/spatial locality optimizations as well as scratchpad memory management (and other forms of software managed memories). To the best of our knowledge, this representation can only support multi-dimensional array references of basic data types (e.g. integers, floats etc.) [8]. Note that scalar variables can follow this definition if we think of them as arrays with zero dimensionality.

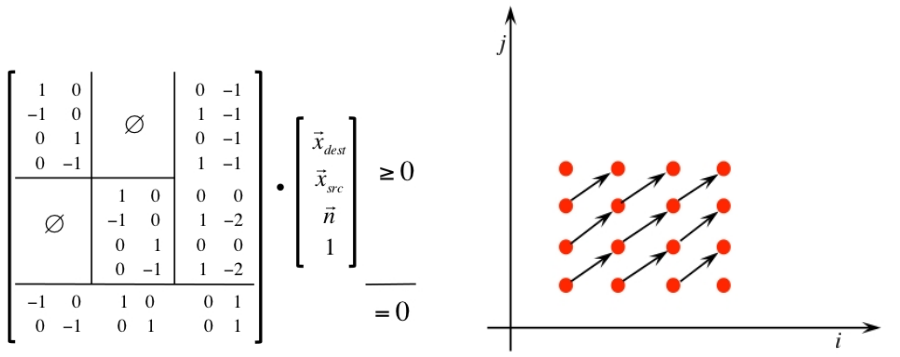
2.4 Polyhedral Dependencies

A vital part of the polyhedral representation of a scop is a set E of polyhedral dependence edges $e \in E$. Polyhedral dependencies are vital because they are the main driving force of a polyhedral analysis engine. Their main difference from conventional dependences is their ability to capture dependence edges between run-time instances of statements thus giving us precise dependence information. A polyhedral dependence $e \in E$ is characterized by a dependence polyhedron P_e consisted of the source and destination execution domains along with an affine transformation that maps each destination (consuming) instance to the corresponding source instance. Figure 2.6 shows an example of a dependence polyhedron representing the true dependence between S_0 and S_1 of Figure 2.5.

2.5 Basic Polyhedral Compilation Flow

The typical polyhedral compilation flow consists of three main stages. The first stage (front-end) is responsible for extracting the polyhedral model from a syntactic representation of the program. The second stage analyses the program and derives a new execution order through affine transformations (i.e. schedules) as described in Section 2.2. Finally, the last stage (back-end) converts the polyhedral model back to a syntactic tree or directly into the target source code. Figure 2.7 depicts a generic polyhedral compilation flow.

There are two kinds of ways a program can enter the polyhedral compilation flow. The user can annotate computation kernels or the entire program could be analysed be the front-end in order to detect maximal program parts that



(a) Dependence polyhedron (b) Dependence edges $(i-1, j-1) \rightarrow (i, j)$

Figure 2.6: This example shows how the true dependence between Π_{00} and Π_{10} of Figure 2.5 is represented in the polyhedral model

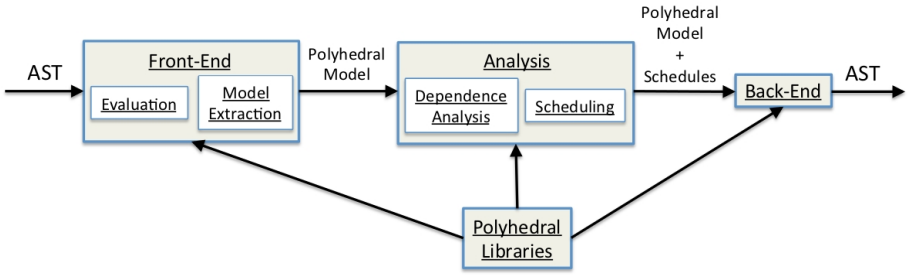


Figure 2.7: Basic polyhedral compilation flow.

satisfy the polyhedral model restrictions. In both cases one needs to evaluate a syntax tree against these restrictions before proceeding to the model extraction phase.

Currently there are two different flavours of polyhedral analysis found in the literature. According to the first approach (fully-automatic) one tries to find an optimal set of affine transformations out of all the legal ones [9, 7, 4] whereas the second approach (semi-automatic) applies individual loop transformations (e.g. loop interchange, loop skewing etc.) and validates the result against the polyhedral dependencies of the program [11, 18]. Figure 3.1 depicts how this generic flow is reflected on an existing polyhedral framework namely PoCC [14].

2.6 Polyhedral Libraries

What really drives the polyhedral compilation engine is a set of library functions providing fundamental abstractions and operations on systems of affine constraints like intersection, union, projection along with integer linear programming solvers (ILP), Fourier-Motzkin elimination and much more. Below is a set of the most commonly used libraries with a small description.

PolyLib Collection of binary and unary operations on vectors, matrices, lattices, polyhedra, Z-Polyhedra and unions of polyhedra. Does not include ILP solvers (`http://icps.u-strasbg.fr/polylib/`).

PIPlib PIP stands for Parametric Integer Programming and is a library used for solving parametric integer linear programs. It is used in order to find the lexicographic minimum (or maximum) in a set of integral points belonging to a convex polyhedron (`http://www.piplib.org/`).

PPL The Parma Polyhedral Library provides abstractions for handling polyhedra along with a rich set of operations. It also provides a parametric integer programming solver based on an exact-arithmetic version of the simplex algorithm (`http://bugseng.com/products/ppl/`).

Omega Similar to PPL and Polylib (`http://www.cs.umd.edu/projects/omega/`).

ISL Inspired by the Omega system, the Integer Set Library provides a complete set of abstractions and operations on systems of affine constraints. It also includes ILP solvers with min/max operations on polyhedra and an intuitive front-end interface/parser system that enables users to utilize the library through a high-level language (`https://www.ohloh.net/p/isl`).

CLooG This library is probably the most vital part of most - if not all - polyhedral frameworks. It provides an extended qulere et al. [15] algorithm for efficiently scanning polyhedra with for-loops. In other words it provides the necessary functionality for converting the polyhedral model back to a syntactic form (`http://www.cloog.org/`).

Chapter 3

Related Work

The first practical and publicly available polyhedral framework for loop-nest optimization was introduced in the mid 90*s* namely *Loopo* [13]. Since its first inception loopo has evolved into a large research compiler framework. It provides a relatively flexible interface through several command-line arguments that can guide the analysis engine and control a rich set of features.

The Pluto compiler is basically an implementation of the Pluto scheduling algorithm [4, 5] with a recent extension to support GPU code generation **??**. It is considered as one of the most robust and powerful scheduling algorithms and can be adjusted through a small set of command-line arguments that control the fusion policy (i.e. maximal fuse, smart fuse and no loop fusion).

The Polyhedral Compiler Collection or in other words the PoCC compiler[1] [14] tries to bring all the advancements of polyhedral compilation under the same roof. It consists of a collection of individual modules (see Figure 3.1) that can be easily extended through a module insertion mechanism.

It uses a standardised interface between the modules which enables PoCC to be integrated into larger compiler infrastructures without changing the modules (i.e. the core functionality). In fact PoCC is integrated to ROSE and can be found under the `projects/polyOpt` directory of the ROSE source tree. The main disadvantage of this approach is that new modules need to be built from scratch without being able to re-use existing functionality already found in neighbouring modules.

The increasing reputation of polyhedral compilation techniques especially after the emergence of heterogeneous parallel architectures like GPGPUs, has attracted attention from the industry as well [17] [2] [3]. We believe that this trend is an indication of the importance of this technology not just for academic purposes but for real-world problems as well.

[1] http://www.cse.ohio-state.edu/~pouchet/software/pocc/
[2] IBM XL Compiler
[3] www.par4all.org

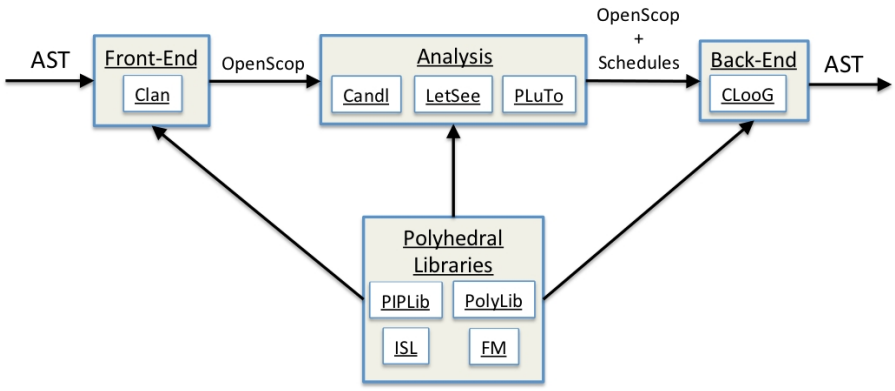


Figure 3.1: The structure of the Polyhedral Compiler Collection reflecting the basic polyhedral compilation flow of Figure 2.7. *Clan* is the front-end module, *Candl* the dependence analysis module and *LetSee* [14] and *PLuTo* [4] are the two currently available scheduling algorithms.

3.1 Motivation

The recurring theme on all polyhedral frameworks mentioned in this chapter is their mission to optimize a computation kernel automatically more or less resembling a black-box. With this work we aim to deviate from this path by creating an API instead of a monolithic compilation flow that serves two main objectives :

Easy to extend and customize Our goal is to enable developers to extend the API easily without having to re-implement core functionality from scratch. We achieve this through a 3-layer interface detailed in Chapter 4. Furthermore, we wish all core functions to be easily customizable as well especially the model extraction conditions and functions. By doing that we hope to encourage users to experiment with this technology without the burden of strict input limitations.

Easy to learn and use Our second goal is to provide a well documented API accessible to a wide-range of users. In particular, we hope to engage new students and researchers to explore the potentials of the polyhedral model without having to deal with low-level concepts and abstractions like matrix layouts, library details/specifications and integer representations. This documentation serves as a first step towards that direction by providing a comprehensive description of the API along with supportive examples and use-cases.

Chapter 4

The RosePolly API

This chapter provides a detailed description of the software design of RosePolly with reference to most class methods and interface functions. This discussion begins with a description of the 3-layer interface followed by a detailed description of each layer. Finally, Section 4.6 shows how one can customize the modeling constraints and functions.

4.1 The 3-Layer Interface

Figure 4.1 depicts the 3-layer interface as a horizontal UML diagram while Figure 4.2 depicts a vertical view of the same concept. The main objective of this design is to separate the three conceptual layers of a polyhedral framework in order to facilitate easy extensibility, maintenance and accessibility for educational and experimental purposes.

4.2 Layer 1–The Modeling/Scheduling Interface

This layer encompasses all the high level functionality of a polyhedral framework which essentially consists of the three main compilation steps of Figure 2.7 namely model extraction (front-end), analysis (mid-end) and code generation (back-end). Of course the interface itself does not impose a compilation flow which is up to the user to specify. The current version of RosePolly provides a default model extraction pass a Pluto scheduling module [4] and a Cloog [3] module for code generation. Notice from Figures 4.1 and 4.2 that one can add analysis or code generation modules by simply extending the `RosePollyModel` class to reflect a new polyhedral policy/mechanism which would re-use all the functionality from layers 2 and 3. Furthermore, the default front-end can be customized according to Section 4.6.

Apart from the `RosePollyModel` class and its inherited classes, the first layer also includes a small yet extensible set of interface methods for user convenience. The rest of this paragraph describes each of these methods along with key

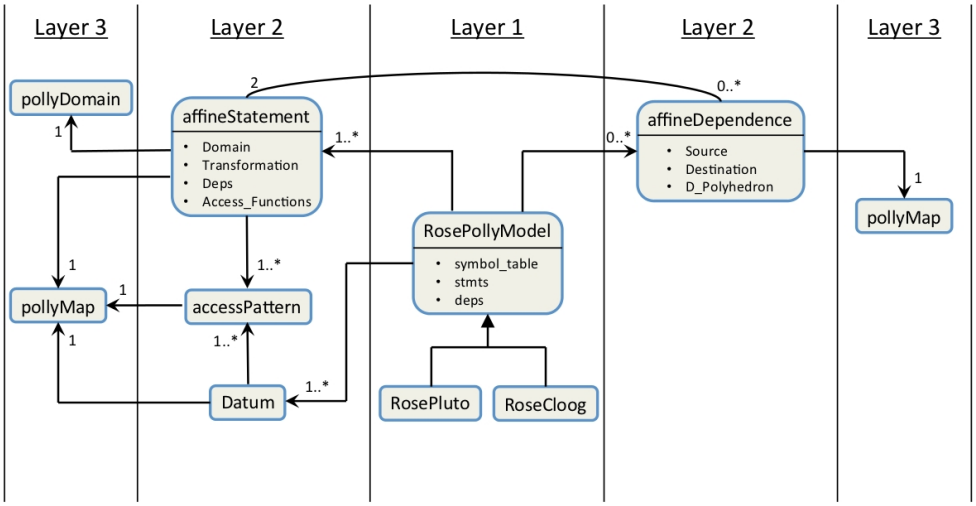


Figure 4.1: A horizontal UML view of the 3-Layer interface of RosePolly

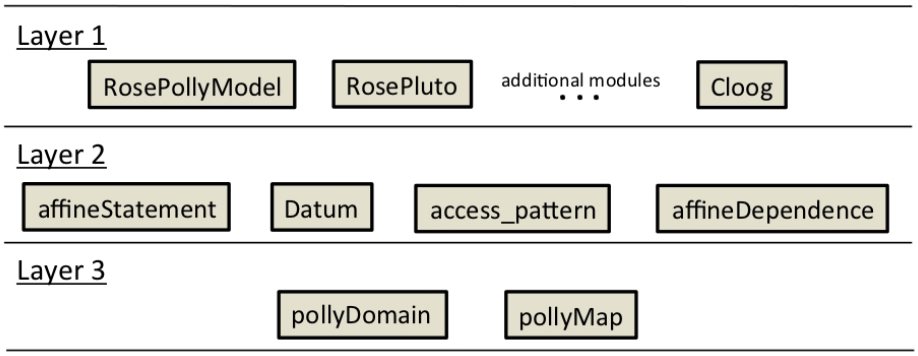


Figure 4.2: A vertical perspective of the 3-Layer interface of RosePolly. Additional user modules can be added to the first layer.

```
#pragma rosePolly
{
  for ( i = 0 ; i < N ; i++ ) {
    for ( j = 0 ; j < N ; j++ ) {
      C[i][j] = 0;
      for ( k = 0 ; k < N ; k++ ) {
        C[i][j] += A[i][k]*B[k][j];
      }
    }
  }
}
```

Figure 4.3: Example of an annotated Matrix-Multiply kernel.

methods from the `RosePollyModel` class, `RosePluto` class and `RoseCloog` class followed by a usage example (Figure **??**) that puts them all together.

4.2.1 The RosePolly Interface

The `RosePollyInterface.h` includes functions for instantiating and destroying polyhedral model objects in order to hide construction and garbage collection details from the user. The five functions currently implemented are the following :

```
vector<RosePollyModel*> RosePollyBuildModel(
                                    SgNode * root,
                                    search_type t=ANNOTATED );
```

This method is essentially the front-end of the polyhedral framework. First of all, It searches a syntactic sub-tree, indicated by the first argument, for candidate computation kernels. This search can be either `ANNOTATED` or `AUTO` according to the second argument which defaults to `ANNOTATED`. If the search is annotated, then the search looks for kernels annotated with a `#pragma` declaration as shown on Figure 4.3. The annotated kernels are then evaluated against the polyhedral restrictions (see Section 4.5 on how these restrictions can be customized) and the valid ones are converted into polyhedral model objects that are returned as an STL vector. On the other hand, the automated search looks for maximal computation kernels that satisfy the polyhedral restrictions and converts them into polyhedral model objects. Its worth noting that if an annotated kernel fails the polyhedral evaluation an informative error message will appear on the screen indicating the cause of the failure.

```
RosePollyModel * RosePollyBuildModel( FlowGraph * graph );
```

This method instantiates a polyhedral model object from a lightweight and robust control flow graph. This graph is first evaluated and then converted into a polyhedral model object. This function is used internally from the `RosePollyBuildModel(SgNode*,search_type)` function but can also be used as a way of re-entering the polyhedral compilation flow after invoking the backend (see `RoseCloog` class).

```
RosePluto * RosePollyBuildPluto( RosePollyModel * model );
```

This method returns a new `RosePluto` object which has a fresh copy of the polyhedral model information from the input polyhedral model object. Note that the user doesnt need to worry about the construction and destruction of these objects.

```
RoseCloog * RosePollyBuildCloog( RosePollyModel * model );
```

This method returns a new `RoseCloog` object which has a fresh copy of the polyhedral model information from the input object.

```
void RosePollyTerminate();
```

This method is responsinble for the garbage collection. This method makes sure that all first-layer objects that have been instantiated through `RosePollyInterface.h` have released their memory resources.

4.2.2 The `RosePollyModel` class

This is the base class for all polyhedral model objects. It contains all the basic information described in Chapter 2. A user can write a new polyhedral module simply by defining a new `RosePollyModel` subclass. Here are some important member methods of the `RosePollyModel` class.

```
polly_iterator<affineStatement> get_statements();
polly_iterator<affineDependence> get_dependencies();
const symbol_table& get_data() const;
int get_id() const;
```

Access methods for the basic Layer 2 information. The `symbol_table` is an associative container currently implemented as an `STL map`, indexed by the symbol of the datum (an `std::string`) and carrying `Datum` objects (see Paragraph 4.4.3). The `polly_iterator` is a basic STL-type container that is used to carry Layer-2 objects.

```
int dep_satisfaction_update( int level );
bool deps_satisfaction_check() const;
```

The first method changes the state of each dependence object to `satisfied` if it is satisfied by the transformation indicated by the integer argument. It returns the number of dependencies satisfied. The second method returns `true` if all dependencies are satisfied and `false` otherwise.

```
void loop_fission();
void loop_skewing( int width, bool top );
```

Methods for aplying loop-fission and loop-skewing transformations to all statements. The loop-skewing method skews `width` number of outer (`top=true`) or inner dimensions (i.e. loop nest levels) for each statement.

```
void print( int ident, bool print_deps ) const;
```

Prints the model information with a given `ident` (defaults to `0`) and can optionally print all the dependencies as well (defaults to `false`).

4.2.3 The `RosePluto` class

This class implements the Pluto scheduling algorithm. It consists of just one public method listed below :

```
apply( fuse_type f );
```

Executes the Pluto scheduling algorithm and infuses the resulting transformations to each statement. The user can specify the fusion strategy by an optional argument that can take tha values `NO_FUSE`, `MAX_FUSE` and `SMART_FUSE` defaulting to the `SMART_FUSE` option.

4.2.4 The `RoseCloog` class

This class implements an interface to the CLooG code generator or in other words it is the default back-end of the RosePolly framework.

```
static CloogOptions * init_default_options() const;
```

Static member of the `RoseCloog` class returning a `CloogOptions` object carrying the default Cloog options.

```
void apply( CloogOptions * opts );
```

Invokes the Cloog code generation algorithm guided by the provided `CloogOptions` object.

```
CloogDomain * get_cloog_domain( pollyDomain * dom ) const;
CloogScattering * get_cloog_scatter( pollyMap * map ) const;
```

Utility methods for creating `CloogDomain` and `CloogScattering` objects from RosePolly Layer-3 objects.

```
FlowGraph * print_to_flow_graph();
void print_to_screen() const;
void print_to_file( const char * name ) const;
```

Various print methods. Note that the `print_to_flow_graph()` method can be used to generate a `FlowGraph` object which we can then use to re-enter the compilation flow through `RosePollyBuildModel(FlowGraph*)`. This is a very useful feature if we wish to implement a feedback-directed compilation flow.

4.3 Layer-1 usage example

```
int main( int argc, char * argv[] ) {

    vector<string> argvList(argv,argv+argc);

    SgProject * proj = frontend(argvList);

    vector<RosePollyModel*> kernels =
            RosePollyBuildModel(proj,ANNOTATED);

    for ( int i = 0 ; i < kernels.size() ; i++ ) {

        RosePluto * pluto = RosePollyBuildPluto( kernels[i] );
        pluto->apply(MAX_FUSE);

        RoseCloog * cloog1 = RosePollyBuildCloog( pluto );

        pluto->apply(SMART_FUSE);

        RoseCloog * cloog2 = RosePollyBuildCloog( pluto );

        CloogOptions * opts = RoseCloog::init_default_options();

        cloog1->apply(opts);
        cloog2->apply(opts);

        /* e.g. degrees of parallelism */
        int metric1 = cloog1->get_metric();
        int metric2 = cloog2->get_metric();

        FlowGraph * graph = (metric1>=metric2) ?
        cloog1->print_to_flow_graph() :
        cloog2->print_to_flow_graph();

        RoseCUDA * cuda = RosePollyBuildCUDA( graph );

        /* to be continued ... */
    }

}
```

4.4 Layer 2–The Polyhedral Model Interface

This layer consists of the notions of a polyhedral statement, a polyhedral dependence and a polyhedral representation of access patterns and data as shown more clearly in the vertical perspective of Figure 4.2.

4.4.1 The `affineStatement` class

Access methods :

```
polly_iterator<affineDependence> get_deps();
pollyDomain * get_domain() const;
pollyMap * get_transformation() const;
polly_iterator<AccessPattern> get_reads();
polly_iterator<AccessPattern> get_writes();
SgExprStatement * get_statement() const;

int get_ID() const;
string get_name() const;
int get_num_reads() const;
int get_num_writes() const;
AccessPattern * get_read( int pos ) const;
AccessPattern * get_write( int pos ) const;
int get_nestL() const;
vector<string> get_iterVector() const;
```

Utility methods :

```
void undo_transformation();
```

Removes the last transformation dimension inserted to the statement. Can be used for feedback-directed analysis.

```
int get_trans_entry( int dim, int pos )  const;
```

This method returns the transformation entry indicated by the coordinates `dim` and `pos` ignoring any scalar dimensions. This is useful when we wish to extract properties like the determinant or the inverse of a transformation matrix without having to deal with scalar dimensions that do not have any effect on the intra-statement execution order (scalar dimensions are used to impose an inter-statement or fussion order only).

```
bool is_in_nest( int * scc, int width )  const;
```

This method returns `true` if a statement is within a specific loop nest indicated by an integer array of ids corresponding to values for the scalar dimensions. The name `scc` stands for Strongly Connected Component and `width` is the size of the array.

```
loop_type detect_parallelism( int level ) const;
```

Returns the loop-type of a transformation dimension as either `PARALLEL`, `PIP_PARALLEL` or `SEQ`. Notice that a statement's dimension being `PARALLEL` doesn't mean that the loop itself will be `PARALLEL` since another statement's dependence could be carried by that loop.

```
void print( int ident ) const;
coid print_deps( int ident ) const;
```

4.4.2 The `affineDependence` class

Access methods :

```
affineStatement * get_src() const;
affineStatement * get_dest() const;
pollyMap * get_polyhedron() const;
```

Utility methods :

```
int src() const;
int dest() const;

bool isValid() const;
bool isSatisfied() const;
int get_satisfaction_level() const;
bool satisfaction_test( int level );

void print( int ident, int * count = NULL, int src = -1, int dest = -1 ) const;
```

The `print` method prints all deps that have a specific source and destination id specified by `src` and `dest`. The default values will print all dependencies. The `count` pointer is used to return the number of dependencies printed. The `src()` and `dest()` methods return the `id` of the source and destination statement respectively.

Finally, each dependence has a type that can be queried with the following methods.

```
bool isR_R() const; /* Read-after-Read   */
bool isW_R() const; /* Read-after-Write  */
bool isR_W() const; /* Write-after-Read  */
bool isW_W() const; /* Write-after-Write */
```

4.4.3 The `Datum` class

This class represents a symbol table entry or in other words a datum accessed by the computation kernel.

Access methods :

```
AccessPattern * get_pattern( int pos ) const;
int num_patterns() const;
string get_name() const;
IO get_IO() const;
bool is_parameter() const;
SgVariableSymbol * get_symbol() const;
Type get_type() const;
```

The `IO` of a `Datum` object can be `IN`, `OUT` or `INOUT` while the type can be `INT`, `FLOAT` or `DOUBLE`.

4.4.4 The `AccessPattern` class

The `AccessPattern` class represents a single memory access operation of the kernel. It's main methods are the following :

```
Datum * get_parent() const;
pollyMap * get_pattern() const;
SgExpression * get_refExp() const;
SgExpression * get_subscript( int pos ) const;

int get_id() const;
string get_name() const;
int get_dim() const;
IO get_IO() const;
bool has_valid_map() const;

void print( int ident ) const;
```

4.5 Layer 3–The Polyhedral Library Interface

The last layer consists of the basic building blocks of a polyhedral model representation which are essentially an interface to the underlying polyhedral library(ies) responsible for the low level storage management, integer representations and mathematical operations like Integer Linear Programming solvers, Fourier-Motzkin elimination and many other unary/binary operations. These building blocks are the Z-Polyhedron embodied by the `pollyDomain` class and the Affine Transformation expressed by the `pollyMap` class.

The user interacts with the contents of these abstractions through a basic interface (Figure 4.5) outlined by the `RosePollyBase` class which is the base class for both `pollyDomain` and `pollyMap` classes. This interface consists of the following methods :

Access interface

```
simple_matrix * get_matrix( bool ineq ) const;
vector<string> get_dims( dim_type t ) const;
int get_total_size() const;
int get_dim_size( dim_type t ) const;
```

The `dim_type` can be `dim_in`, `dim_out` or `dim_param`.

Modifiers

```
void set_matrix( simple_matrix * mat, bool ineq );
void set_constraints( simple_matrix * ineq, simple_matrix * eq );
void append_inequalities( simple_matrix * mat );

void add_dim( dim_type t, string name );
```

The vehicle for this interaction is the `simple_matrix` class which is a light-weight matrix abstraction with the following basic methods :

```
void set_entry( int i, int j, int value );
int entry( int i, int j ) const;

int get_rows() const;
int get_columns() const;

void append( const simple_matrix& m );
void swap_rows( int r1, int r2 );
void remove_column( int pos );
void add_zero_row( int pos );
```

Figure 4.5 shows the column layout of a `simple_matrix` for a `pollyDomain` and a `pollyMap` object respectivelly.

Unfortunately, the `simple_matrix` abstraction is not powerful enough to capture rational expressions that can be found in some programs. If the user wants a more low-level manipulation the underlying library abstraction can also be accessed through the following methods :

```
integer_set get_integer_set();
integer_map get_integer_map();
```

The `integer_set` and `integer_map` types are mapped to the corresponding library types in the `RosePollyMath.h` header. The library used in the current version of `RosePolly` is the `ISL` library (see Section 2.6) which means that the `integer_set` and `integer_map` types can be used as `isl_set` and `isl_map` types respectively in order to leverage the reach features of `ISL`.

| dim_in | params | cst |

(a) pollyDomain

| dim_in | dim_out | params | cst |

(b) pollyMap

Figure 4.4: Column layout for pollyDomain and pollyMap objects

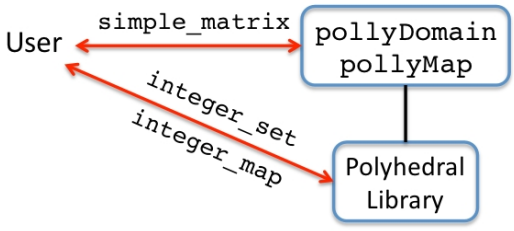


Figure 4.5: The contents of pollyDomain and pollyMap objects can be accessed through a light-weight simple_matrix abstraction or by accessing the underlying library abstraction for low-level manipulation

4.5.1 The pollyDomain and pollyMap classes

The following methods are currently supported by both classes :

```
void subtract( pollyDomain * d );
void intersect_domain( pollyDomain * d );
void intersect_range( pollyDomain * d );

void lexmin( bool non_neg );
void lexmax();
bool is_empty() const;
```

The void intersect_range(pollyDomain*) method is only meaningfull for a pollyMap object since it performs an intersection operation between the output domain of the pollyMap object and d. In case of a pollyDomain object, this method has no effect.

The lexmin method has the option of non-negative minima.

Pretty print methods are also provided :

```
void print( int ident ) const;
void print_matrix( int ident, bool ineq ) const;
```

The first print method uses the underlying library functionality (if any) to print the corresponding Layer-3 object while the second print_matrix method prints the Layer-3 object as a simple_matrix with the option of choosing between the equality and inequality constraints.

4.6 Customizing the Modeling Constraints and Functions

One of the key feature of the RosePolly framework is the ability to customize the polyhedral modeling constraints and extraction functions. This can be done by providing an additional argument to the RosePollyBuildModel methods of Paragraph 4.2.1.

```
vector<RosePollyModel*> RosePollyModelBuild(
                                SgNode * root,
                                RosePollyCustom * custom
                                search_type t );
```

Or :

```
RosePollyModel * RosePollyBuildModel(
                                FlowGraph * graph,
                                RosePollyCustom * custom );
```

In order to use these methods one needs to implement the RosePollyCustom interface which is the following :

```
class RosePollyCustom {

    public:
    virtual bool evaluate_loop( ForLoop * loop ) =0;
    virtual pollyDomain * add_loop( pollyDomain * d,
                                ForLoop * loop ) const=0;

    virtual bool evaluate_conditional(
                            Conditional * cond ) const=0;
    virtual pollyDomain * add_conditional(
           pollyDomain * d, Conditional * cond ) const=0;

    virtual bool evaluate_access(
                            AccessPattern * ap ) const=0;
    virtual pollyMap * add_pattern( pollyDomain * m,
                            AccessPattern * ap ) const=0;

    virtual void add_params( vector<string> p ) =0;
}
```

The evaluate_loop(ForLoop*) method can change the state of the calling object because if the evaluation succeeds then the evaluation of the inner-most loops need to take the outer-most induction variables into account.

The add_params method handles the minimum information provided to a RosePollyCustom object by the RosePolly runtime.

For user convinience the following utility functions are also provided as protected members of the RosePollyCustom class :

```
bool isAffineExpression( SgExpression * exp,
                         vector<string>& legalVars );
bool isIntExpression( SgExpression * exp );
```

The first one returns `true` if a given `SAGE III` expression is affine according to a set of legal symbols, while the second returns `true` if an expression can be evaluated as an integer value at compile-time.

The current version of the framework has a default implementation of the `RosePollyCustom` interface namely `defaultRosePollyCustom` which can be used as a base in order to avoid implementing the whole set of evaluation and extraction methods.

The `ForLoop` and `Conditional` objects are internal abstractions for the for-loop and conditional respectively. Their construction and destruction is handled by the framework. The construction of `ForLoop` objects comes with a set of strict syntactic restrictions that have nothing to do with the polyhedral model but with the early development version of the framework. These constraints might be completely eliminated in future versions and are :

Strict Loop constraints

- One induction variable.
- One initialization statement
- Possitive unit stride written with `++` operator.
- Single variable on the LHS of condition that matches the induction and initialized variable.

Here are some basic Access methods for the `ForLoop` and `Conditional` classes :

`ForLoop` **class**

```
SgExpression * get_start() const;
SgExpression * get_end() const;
SgForStatement * get_loop() const;

string unparse_start() const;
string unparse_end() const;
string get_symbol() const;
```

`Conditional` **class**

```
SgExpression * get_exp() const;
```

In other words, this is the set of access methods the user can use within a `RosePollyCustom` object in order to customize the polyhedral evaluation and modeling functions.

Keep in mind that `ForLoop` and `Conditional` objects carry a type that can be either head or tail. This can be queried through the method :

```
bool is_head() const;
```

and the user is responsible for handling both cases (i.e. loop-head, conditional-head, loop-tail and conditional-tail) for each node type.

References

[1] *Optimizing Compilers for Modern Architectures : A Dependence-Based Approach / R. Allen, K. Kennedy.* San Francisco, EUA : Morgan Kaufmann, 2002. 5

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. 5

[3] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, 2004. 5, 13

[4] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 132–146. Springer Berlin / Heidelberg, 2008. 3, 9, 11, 12, 13

[5] U. Bondhugula and J. Ramanujam. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical report, 2007. 3, 11

[6] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Springer, 2000. 3

[7] A. Darte and F. Vivien. Automatic parallelization based on multi-dimensional scheduling, 1994. 9

[8] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20:23–53, 1991. 8

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International Journal of Parallel Programming*, 21:313–347, 1992. 3, 9

[10] P. Feautrier. Automatic parallelization in the polytope model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*,

volume 1132 of *Lecture Notes in Computer Science*, pages 79–103. Springer Berlin / Heidelberg, 1996. 3

[11] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34:261–317, 2006. 2, 9

[12] T. Grosser. Enabling Polyhedral Optimizations in LLVM. Technical report, University of Passau, 2011. 2

[13] C. Lengauer. Loop parallelization in the polytope model. In *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 398–416. Springer Berlin / Heidelberg, 1993. 11

[14] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Conference on Supercomputing (SC'10)*, New Orleans, LA, Nov. 2010. 9, 11, 12

[15] F. Quiller, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28:469–498, 2000. 10

[16] J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing*, 9:365–389, 1995. 10.1007/BF01206273. 5

[17] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *GCC Research Opportunities Workshop (GROW'10)*, Pisa, Italie, Jan. 2010. 11

[18] N. Vasilache, C. Bastoul, A. Cohen, and S. Girbal. Violated dependence analysis. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 335–344. ACM, 2006. 9

[19] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA, 1990. 5