ATMOL: A Domain-Specific Language for Atmospheric Modeling

Robert A. van Engelen

Department of Computer Science, Florida State University, FL32306, USA

This paper describes the design and implementation of ATMOL: a domain-specific language for the formulation and implementation of atmospheric models. ATMOL was developed in close collaboration with meteorologists at the Royal Netherlands Meteorological Institute (KNMI) to ensure ease of use, concise notation, and the adoption of common notational conventions. ATMOL's expressiveness allows the formulation of high-level and low-level model details as language constructs for problem refinement and code synthesis. The atmospheric models specified in ATMOL are translated into efficient numerical codes with CTADEL, a tool for symbolic manipulation and code synthesis.

Keywords: scientific computation, code generation, problem-solving environments, high-performance computing

1. Introduction

The simulation of realistic atmospheric processes is computationally intensive. A typical weather forecast for the next day, for example, requires about a trillion (10^{12}) arithmetic operations. Even with the immense processing power of today's supercomputers, a short-term weather forecast can take hours to complete on a high-performance machine. Atmospheric models, such as climate models, ocean circulation models, and numerical weather forecast models are notorious for their demand for computing power. These scientific applications make a trade-off between the accuracy of the numerical solution and the maximum amount of computing time that can be alloted to produce a solution.

Because processing power has significantly increased by the development of new high-performance architectures such as massive SMP machines, this has lead to new opportunities for improving atmospheric models. In particular, the discretization and solution methods can be improved and grid resolutions can be increased. However, improvements in the model formulation and solution methods require significant programming efforts, because a simple "plug-and-play" development paradigm with software components [24] does not yet exist for scientific applications.

In addition, it is difficult to develop new scientific software for high-performance machines or to port existing scientific software to these machines [21]. This is mainly due to the shortage and weakness of available development tools. Even the most advanced restructuring and parallelizing compilers cannot effectively restructure low-level source codes, see e.g. [6, 10, 20, 25].

Several software tools and problem-solving environments (PSEs) [16] have been built that aid the development of applications for solving scientific problems [4]. Most PSEs do not actually generate code but primarily offer an environment for simulation. Well-known examples of these are ELLPACK [22], its parallel version //ELLPACK [17], and various simulation tools programmed in MATLAB [18]. The computational kernels of these PSEs consist of a large library of routines containing many numerical solution methods.

This paper describes the design and implementation of ATMOL: a domain-specific language for the formulation and implementation of atmospheric models. ATMOL is translated and compiled into efficient numerical codes with CTADEL [29, 31, 27]. Code synthesis tools such as DEQSOL [26], ALPAL [12], SCINAPSE [2], and

CTADEL generate code from higher-level specifications of PDE-based models. The numerical knowledge of these systems is determined by the expressiveness of the problem specification language and the underlying translation techniques. The many advantages of code synthesis from a higher-level specification are summarized below.

Increased Productivity

One of our design goals for ATMOL was to ease the task of developing new codes for atmospheric models and to alleviate the burden of maintaining these codes. A "plug-and-play" implementation with software components and libraries does not exist in the field of scientific computing yet, although several attempts have been made at this, see e.g. [11]. Therefore, it is quite common that the simulation code of a model is written almost entirely from scratch. Also, the model and its simulation code require several design and implementation iterations before the code can be employed in a production environment. In each iteration, the code is modified or completely rewritten depending on the improvements. Because the modification of previous versions of code is error prone and writing new codes for each model improvement is prohibitively expensive, automated code synthesis is a valuable approach that could increase the productivity of implementing atmospheric models. For example, code synthesis of the "dynamics" part of the HIRLAM model [9] took about one hour. This is extremely fast compared to the original implementation of this code by hand which took several months. In addition, the generated codes outperformed the original hand-written codes on several high-performance machines [29]. The specification of a model requires some effort when a user is not familiar with ATMOL. However, the effort to write a specification of a larger model such as a weather forecast model in ATMOL is not nearly as high as writing the full numerical code.

Enhanced Maintainability

Scientific software is subject to a lot of changes during its lifetime. New methods and techniques are frequently added and existing numerical solution methods are improved over time. This requires code maintenance, which is greatly alleviated by automatic code synthesis. Whenever changes are made to the model description or solution methods, a new code can be synthesized that reflects these changes. However, this assumes that the specification language is powerful enough to enable these changes to be made to the model description without too much effort.

Increased Reliability

The construction of numerical models and their codes by hand involves the use of certain translations that are based on mathematical principles. We formalized these rules and implemented them as transformation rules in a term rewriting system which drives CTADEL's code synthesis. For example, synthesis of the HIRLAM "dynamics" code requires the application of these rules hundreds of thousands of times. We have verified that the individual translation rules are correct. Therefore, the resulting synthesized code can be assumed to be correct if the model specification is correct. This had a major impact on reliability of the model and its code. After comparing the synthesized codes for the HIRLAM system with the original hand-written production HIRLAM code, we discovered the differences which were due to several errors made by programmers in the original hand-written code. One error was a programming mistake related to updating the values at the boundaries of certain array variables. Another error was found in the numerical scheme: the forecast model can deal with spherical grids only, while the model was originally intended to handle more general curvi-linear grids. A third problem was found in the model description involving the wrong units of dimensionality (the hand-written code was correct in this respect) and the model formulation was corrected accordingly. Also, we found that certain assumptions were made in the original hand-written code which have not been documented as requirements for the correct use of the code. By coincidence, these mistakes did not affect quality of the forecast (a case of luck rather than wisdom). Based on these experiences, we believe that formal methods, and in particular code synthesis from domain-specific abstract problem specifications, have great advantages in terms of correctness, reliability, and consistency of model codes and documentation.

Flexibility

ATMOL's extensibility is essential, because it is impossible to anticipate new model implementation developments that use specialized and application-specific solution methods. These methods may be substantially different from those used in the specification of other models, e.g. weather forecast models. Also, a code synthesis system with hardwired language constructs hampers maintainability when new operators, methods, and algorithms cannot be added to the system.

The remaining part of this paper is organized as follows. In Section 2 we present the atmospheric modeling language ATMOL and we illustrate its special features using an example atmospheric model. Section 3 discusses the design and implementation of ATMOL and its uniform notational conventions for aggregate operations from which CTADEL derives its power to translate high-level specification into low-level program constructs. Finally, some concluding remarks are given in Section 4.

2. ATMOL

The ATmospheric MOdeling Language (ATMOL) was developed in close collaboration with meteorologists at the Royal Netherlands Meteorological Institute (KNMI) to ensure the ease of use, concise notation, and the adoptation of common notational conventions. The high-level constructs in ATMOL are *declarative* and *side-effect free* which is required for the application of transformations to translate and optimize the intermediate stages of the model and its code. ATMOL is *strict* and requires the typing of objects before they are used. Three different type systems are used to check the model and to pinpoint problems with the specification at an early stage. In this section we will introduce PDE-based scientific models, describe the specification of those models in ATMOL, and present example specifications.

2.1. PDE-Based Models

A scientific model can be written generally as a system of n time-dependent (coupled) PDEs of the form

$$\frac{\partial}{\partial t}\mathcal{L}_i(u_i) = F_i(u_1, \ldots, u_n) \quad i = 1, \ldots, n, \tag{1}$$

together with a set of *initial conditions* and a set of *boundary conditions* which are said to hold on the boundary $\partial\Omega$ of the domain $\Omega \subseteq \mathbb{R}^d$ (or $\mathbb{C}^d$) with dimension d. Here, $u_i = u_i(\mathbf{x}, t)$, $i = 1, \ldots, n$, are called the *dependent variables* of the PDE problem which are functions of the space coordinates $\mathbf{x} \in \Omega$, and time t. The space coordinates $\mathbf{x}$ and time t are called the *independent variables* of the PDE problem. Variables are often referred to as *fields* in association with mapping them on grids. In the sequel we will use the phrase *variable* and *field* interchangeably. F_i is a function involving the u_i as well as their space and time derivatives and $\mathcal{L}_i$ is a space differential operator which in most cases is the identity operator, i.e., $\mathcal{L}_i(u_i) = u_i$, or the Laplacian operator, i.e., $\mathcal{L}_i = \Delta = \nabla^2$. So-called *steady-state* problems are time-independent problems in which the time derivative $\frac{\partial}{\partial t}$ on the left-hand side of Eq. (1) is removed.

2.2. The Specification of the Model Variables in ATMOL

A model specification starts with the declaration of independent variables. These are declared with the construct

`space (`*vars*`)` [`time` *var*]`.`

where *vars* is a comma-separated list of spatial coordinates with optional index parameters and *var* is the time variable. The time variable declaration can be omitted[1] to specify a steady state problem. For example

```
space (x(i),y(j),z(k)) time t.
```

declares a three-dimensional model space with spatial coordinates x, y, and z, discretized on a grid indexed by variables i, j, and k.

Dependent variables of a model are declared with the construct

[1] We will use the informal meta-notation [ ] to denote optional constructs throughout this paper.

var `::` [*type*[(*lb*`..`*ub*)] [`dim "`*unit*`"`]] [`field` *coord*] [`monotonic` *mono*] [`on` *domain*].

This declares a dependent variable *var* with its model-specific properties. The *type* of a variable is either `integer`, `float`, `complex`, or `boolean`, with `float` the default. The bounds *lb* and *ub* optionally specify the lower bound and upper bound on the values for the variable. The *unit* of a variable is a string that specifies the dimensional unit, which can be expressed as SI-units and the most common derivative units and the "`*`", "`/`" and "`^`" operators (e.g. `"km/s^2"` and `"J/kg/K"`). The *coord* part specifies the dependencies of the variable on the independent variables (spatial coordinates) and also the type of grid for each spatial dimension is specified. The optional *mono* part declares monotonicity properties of the variable in one or more spatial dimensions. The optional *domain* specifies the discretized grid domain of the variable.

A *scalar variable* is declared with only a type and an optional unit. For example,

```
r :: float dim "m".
```

A *field* is a dependent variable that is mapped on a grid. It is specified with spatial coordinates and a discrete grid domain. For example, the first component of the velocity vector field of the HIRLAM weather forecast model [9] is declared as

```
u :: float dim "m/s"
     field (x(half),y(grid),z(grid))
        on i=1..n by j=1..m by k=1..l.
```

where the `x`, `y`, and `z` coordinates span the three dimensional atmospheric space and the `grid` and `half` coordinate annotations specify the type of grid in each dimension. Value range and monotonicity information of a variable can be specified. For example, the pressure field that can be found in the HIRLAM weather forecast model is declared as

```
p :: float(0..107000) dim "Pa"
 field (x(grid),y(grid),z(grid))
   monotonic k(+) on i=1..n by j=1..m by k=1..l.
```

The pressure variable p can range between 0 and 107000 and increases with the increasing vertical grid index `k` (`k` runs towards the earth surface). Monotonicity information is useful to derive efficient code for certain search functions on grids [28]. The specification of value ranges is important for the optimization of conditional expressions that implement boundary conditions.

2.3. The Specification of the Model Equations in ATMOL

The notational conventions adopted by ATMOL allows the PDEs of an atmospheric model to be specified in concise vector notation. Each equation of the set of coupled PDEs of a model is defined using the infix "`=`" operator. The PDE right-hand sides are arithmetic expressions that can use any of the PDE operators shown in Table 1. The boundary conditions are given as conditional expressions of the form "E `if` L", where E is an expression and L is a logical expression.

Operator	Description	Alternative Notation
`grad` E	Gradient ∇E	`nabla *` E
`div` E	Divergence $\nabla \cdot E$	`nabla .*` E
`curl` E	Curl $\nabla \times E$	`nabla #*` E
`lapl` E	Laplacian $\nabla^2 E$	`nabla^2 *` E
`d` E`/d` V	Partial derivative $\frac{\partial E}{\partial V}$	
`int(`E`,` B`)`	Integration $\int_B E$	
`.*`	Dot product $\cdot$	
`#*`	Cross product $\times$	

Table 1. PDE Operators.

Macros can be defined for convenience in ATMOL using the "`:=`" operator. Macros are also used for specifying the components of vectors necessary for the formulation of a model in vector notation. The automatic translation of the equations to scalar form requires the application of the chain-rule to compute symbolic derivatives. The chain-rule depends on the coordinate system used by a model. The coordinate system is set by assigning the `coordinates` and `coefficients` macro vectors. For example, the macro definitions

```
coordinates := [x, y]; coefficients := [1, 1].
```

set a two-dimensional Cartesian coordinate system for the model.

2.4. Declaration of PDE Operators in ATMOL

New PDE operators can be added to ATMOL. The syntax of an operator declaration is

op `::` [*type*[(*lb*..*ub*)]] [`dim` *unit*]] [`field` *coords*] [`:=`*E*] .

The operator *op* is a functional of the form $F(E_1, \ldots, E_n)$. Declarations of the arguments E are of the form

arg `::` [*type*[(*var*..*var*)] [`dim` *unit*]] [`field` *coords*]

where *arg* is a variable name, a wildcard "_", or a symbolic expression that serves as a pattern. The *type*s are optional and can be any existing type, including a type variable. When omitted, the type of the arguments are assumed to be the same as the result type of operator. The bounds *lb* and *ub* specify the lower bound and upper bound of the return value of the operator, which can be a symbolic expression using the variables that are part of the value ranges of the arguments expressed with variables *var*. . .*var*. The *unit* of a variable is a string that specifies the dimensional unit expressed in SI-units and most common derivatives of an expression that calculates the unit from the units of the arguments. The *coord* part specifies dependencies of the variable on the independent variables (coordinates) and the type of grid.

Figure 1 depicts the declarations of pre-defined and most commonly used PDE-operators: derivatives, differences, integrals, and midpoint quadratures. The derivative operator "`df`" is the "inert" variant of the $\mathtt{d}E/\mathtt{d}V$ partial derivitive operator. The latter operator applies symbolic derivation using the chain-rule to obtain the derivative of E, which results in a form that uses the inert "`df`" operator. Notice that `U1` and `U2` are unit variables and `U1/U2` denotes the resulting dimensional unit of the "`df`" operator. The "`df`" derivative operator is overloaded and will be replaced by a concrete operator in the equations. In this case the concrete operators are the finite difference operators "`df_g`", "`df_h`", and "`df_c`" that implement three different centered differences depending on the grid location (at grid points or at half points). The implementation of `df_h` is shown in Figure 1. Similarly, the integral "`int`" operator is overloaded and has three different concrete implementations. The first two are midpoint quadratues and the third is the trapezoidal quadrature.

```
df(_ :: float dim U1, _ :: coordinate dim U2)
  :: float dim (U1/U2).
df(_, _)‘grid_overloaded := [df_g, df_h, df_c].
df_h(_ :: field x(half), x) :: field x(grid).
df_h(_ :: field y(half), y) :: field y(grid).
df_h(_ :: field z(half), z) :: field z(grid).
df_h(E :: float dim U1, X :: coordinate dim U2)
  :: float dim (U1/U2) := (E-E@(X=X-1))/delta X.
int(_ :: float dim U1, _ :: domain(coordinate) dim U2)
  :: float dim (U1*U2).
int(_, _)‘grid_overloaded := [mid_g, mid_h, trap].
mid_h(_ :: field x(half), x=_..._) :: field x(grid).
mid_h(_ :: field y(half), y=_..._) :: field y(grid).
mid_h(_ :: field z(half), z=_..._) :: field z(grid).
mid_h(E :: float, X = L .. U :: domain(coordinate))
  :: float := sum(E * delta X, X=L..U-1).
```

Fig. 1. Derivatives, Differences, Integrals, and Quadratures.

2.5. Specification of the HIRLAM Model

In this section, we illustrate the ATMOL specification language with realistic examples from the "dynamics" and "physics" parts of the HIRLAM weather forecast system. Due to limitations in space, only the essential subsets of the "dynamics" and "physics" will be shown. The specification and code synthesis of the HIRLAM "dynamics" and "physics" can be found in [27].

The atmospheric models adopted by climate and weather forecast systems are characterized by two main computational components: the "*dynamics*" and the "*physics*". The "dynamics" is primarily concerned with the fluid dynamics of the atmosphere, while the "physics" is concerned with the computation of physical parameterizations. In the "dynamics" part of a weather forecast system, the *Primitive Equations* [14], which describe the behavior of the prognostic variables, are solved. In the "physics" part, the aggregate effect of subgrid processes and of the processes not described by the PDEs of the primitive equations is computed, such as the effect of solar radiation on the atmosphere. Both the "dynamics" and "physics" operate on the same three-dimensional grid that contains a simulated part of the atmosphere.

Examples of climate and weather forecast systems with the structure described above are: the HIRLAM numerical weather forecast system developed by the HIRLAM project group [7, 9], the IFS model of ECMWF and its parallelized version [8], the CCM2 Community Climate Model

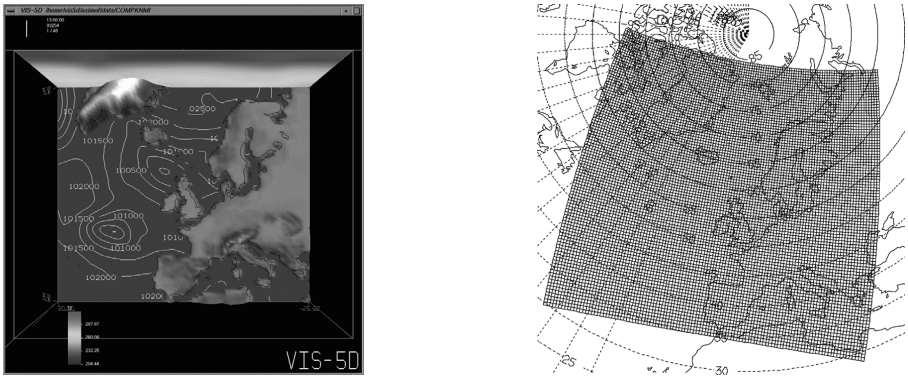

Fig. 2. HIRLAM Weather Forecast and Grid Mapping.

developed by the National Center for Atmospheric Research (NCAR) [5], and the CSIRO Atmospheric General Circulation Model developed by the CSIRO Division of Atmospheric Research [19].

The HIRLAM (HIgh Resolution Limited Area Model) system [9] is a production code written in Fortran 77. The HIRLAM system comprises a so-called "limited area model" that encompasses a local part of the globe which is logically of rectangular shape, see Figure 2 (forecast shown with Vis-5D [15]). The "dynamics" of the HIRLAM system is one of the most computationally intensive numerical routines. In the "dynamics" a set of three-dimensional coupled non-linear hyperbolic partial differential equations is solved.

2.5.1. Dynamics

The *surface pressure tendency equation* and the *equation for the auxiliary horizontal wind velocity vector field* of the HIRLAM "dynamics" [9] are respectively

$$\frac{\partial p_s}{\partial t} = -\int_0^1 \nabla \cdot \mathbf{V}\, dz \qquad (2)$$

$$\mathbf{V} = \begin{pmatrix} u \\ v \end{pmatrix} \frac{\partial p}{\partial z} \qquad (3)$$

Eq. (2) describes the tendency of the surface pressure p_s and Eq. (3) describes the auxiliary horizontal wind velocity vector field $\mathbf{V}$ with two components u_{aux} and v_{aux}. These PDEs are discretized by using finite differences on a logically rectangular grid.

Arrangements of variables in the horizontal domain of atmospheric models were first classified by Arakawa [3]. The u, v, and p fields are arranged on a staggered Arakawa-C grid as depicted in Figure 3. Variable p_s is located on the staggered grid at the p points, u_{aux} and v_{aux} are located at the u and v points, respectively.

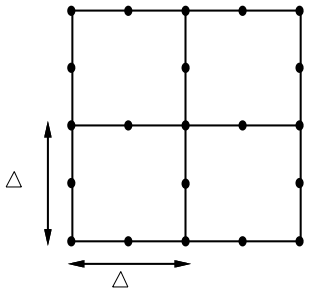

Fig. 3. Horizontal Arrangement of Variables on an Arakawa-C Grid.

The first line of the specification of the "dynamics" in ATMOL shown in Figure 4 declares the x, y, z, and t independent variables and the index variables `i`, `j`, and `k` that span the entire discrete region of the domain Ω of the model. Variables `n`, `m`, and `l` are scalar positive integers which hold the grid size in the longitudinal, latitudinal, and vertical directions, respectively. The vertical grid is at least two points high, which is a useful information for optimal code synthesis. This ensures that the lower and upper boundary conditions of the model do not apply simultaneously. Next, two macros are defined for use in field declarations: `atmosphere` spans the entire grid domain and `surface` spans the horizontal grid domain on the surface. Finally, model fields are declared followed by the two

```
% Declare spatial and time dimensions:
space (x(i),y(j),z(k)) time t.
% Declare grid size variables n, m, and l:
n :: integer(1..infinity); m :: integer(1..infinity); l :: integer(2..infinity).
% For convenience, define macros for two grid domains spanning (i,j,k):
atmosphere := i=1..n by j=1..m by k=1..l; surface := i=1..n by j=1..m.
% Set coordinate system for symbolic derivation with chain-rule:
coordinates := [x, y]; coefficients := [h_x, h_y].
% Declare the model fields:
u     :: float dim "m/s" field (x(half),y(grid),z(grid)) on atmosphere.
v     :: float dim "m/s" field (x(grid),y(half),z(grid)) on atmosphere.
u_aux :: float dim "Pa m/s" field (x(half),y(grid),z(half)) on atmosphere.
v_aux :: float dim "Pa m/s" field (x(grid),y(half),z(half)) on atmosphere.
p     :: float(0..107000) dim "Pa" field(x(grid),y(grid),z(grid)) monotonic k(+) on atmosphere.
p_s_t :: float dim "Pa/s" field (x(grid),y(grid)) on surface.
% Define macro for the horizontal wind velocity vector components:
V := [u_aux, v_aux].
% Equations:
p_s_t = -int(nabla .* V, z=1..l).   % Eq. (2)
    V = [u, v] * d p/d z.           % Eq. (3)
```

Fig. 4. Specification of the Surface Pressure Tendency in ATMOL.

equations. Field declarations with the Arakawa-C grid arrangements shown in Figure 3 and the equations of the submodel, are specified in CTADEL as depicted in Figure 4. The `half` and `grid` grid types are declared in ATMOL and are available as pre-defined types.

2.5.2. Physics

In the so-called cloud routine of the HIRLAM "physics" [9], three layers of clouds are calculated from the "cloudiness" where the layers are defined between four air pressure (p) levels: surface pressure $p = p_s$, $p = 200$ hPa, $p = 500$ hPa, and $p = 800$ hPa, respectively, see Figure 5. In HIRLAM, the vertical coordinate of the global three-dimensional coordinate system is determined by levels of equal air pressure. For each pressure layer, the maximum value in the vertical direction (z) of the cloudiness parameter (c) defines the low cloud (cl), medium cloud (cm), and high cloud (ch) parameters. Vertical ranges of the layers vary with horizontal position because the air pressure p varies.

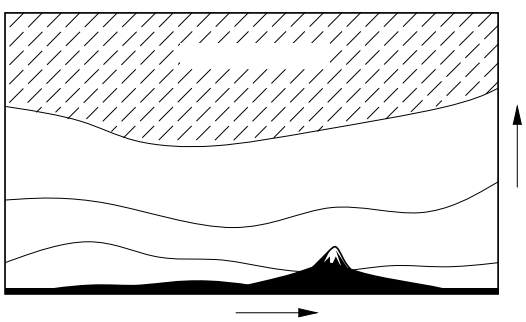

Fig. 5. Schematic Illustration of Three Cloud Layers.

The specification in ATMOL is shown in Figure 6. For this model, the two-dimensional horizontal domain has collapsed to one dimension consisting of $nm = n \times m$ grid points, which is consistent with the implementation of the physics in HIRLAM. The `maxval` operator is a reduction operator that returns the maximum value of an expression on a grid, here for `k=1..l` (see also Table 3 in Section 3.4). More details can be found in [28].

3. Design and Implementation

In this section we will discuss ATMOL's design choices and their implementation. The expressiveness of ATMOL allows specification of most atmospheric models. However, ATMOL is not limited to high-level model specifications and allows the specification of lower-level constructs such as discretizations and Fortran-like program code for the numerical solution algorithms. New user-defined PDE-based operations can be defined, new aggregate operations can be added and their implementation can be described by a procedural algorithm in Fortran-like code. ATMOL is basically a transformation

```
% Declare spatial (collapsed to one-dimension) and time dimensions:
space (xy(ij),z(k)) time t.
% Declare grid size variables nm and l:
nm :: integer(1..infinity); l :: integer(2..infinity).
% Declare the model fields:
p  :: float(0..107000) dim "Pa" field monotonic k(+) on ij=1..nm.
c  :: float on ij=1..nm by k=1..l.
cl :: float on ij=1..nm.
cm :: float on ij=1..nm.
ch :: float on ij=1..nm.
% Equations:
cl = max(maxval(c if p>80000,              k=1..l), -1).
cm = max(maxval(c if p>50000 and p<=80000, k=1..l), -1).
ch = max(maxval(c if p>20000 and p<=50000, k=1..l), -1).
```

Fig. 6. Determination of Cloud Layers in ATMOL.

language with which the operational semantics of the model operators is defined. In fact, all built-in PDE-operators, solution methods and algorithms are written in ATMOL and provided as pre-defined constructs.

3.1. Levels of Abstraction

ATMOL language constructs can be distinguished at five different levels of abstraction and they present a *separation of concerns* for the implementation of a model. From the highest to the lowest level these are, respectively:

Meta-level programming for *symbolic manipulation* of *algebraic expressions* is one of the key technologies in mathematical software. The user interacts with CTADEL by issuing commands to symbolically translate ATMOL constructs.

Model declarations are specified in ATMOL, as described in the previous sections.

A coordinate-free scalar PDE problem is obtained from the model declaration by converting vector notation into scalar notation.

The numerical schemes are obtained by application of the grid type system to coerce continues operators such as partial derivative into discrete operators such as finite differences. Further symbolic manipulation on the results yield optimized numerical solution schemes involving array variables instead of model fields.

Program code is generated from the optimized schemes.

ATMOL allows specification of a model at mixed abstraction levels. This is useful for models that incorporate lower-level non-PDE-based operators, such as the operations in the typical "physics" part of an atmospheric model. A mixed high- and low-level specification can also be used to bypass CTADEL's automatic translations for (parts of) the model. This gives a user more control over the implementation of the synthesized solutions by allowing replacement of higher-level constructs with lower-level constructs in the model declaration. CTADEL's translation is deterministic, but optimal translation results are not unique due to the nature of the problem at hand. In some cases, the translation process may produce an intermediate optimal solution that the user wants to modify to match certain requirements, e.g. based on numerical properties.

Model descriptions are translated into code through several problem refinement stages. Refinement takes place by the application of commands to create new intermediate solutions. These commands can be run automatically in a script or they can be applied by a user. Commands are available for every refinement stage: *type checking* and coercion of the model equations and fields, *unit analysis* to verify the consistency of the model, *grid conversion* and *discretization* of the continuous equations by second-order finite differences and replacement of integrations with midpoint quadratures, *simplify* the discrete equations, determine *array bounds* of discretized variables, *eliminate common-subexpressions* and optimize the intermediate code for a *target computer architecture*, *generate code* in Fortran 77, data-parallel For-

tran 90, HPF (High-Performance Fortran), or parallel code in Fortran 77 with MPI [13]. To generate distributed parallel code in Fortran 77 with MPI, CTADEL implements a domain splitting technique for parallelizing the model [30, 27].

3.2. Expression Syntax for PDEs, Intermediate Constructs, and Program Codes

The syntax of ATMOL is defined as an *operator precedence grammar* [1] which provides a simple and convenient mechanism to dynamically extend the syntax, e.g. while processing problem specification source files. A single operator precedence grammar is used by the CTADEL system which defines the basic grammar for symbolic expressions E:

$$\begin{aligned} E &\rightarrow V \mid C \mid \oplus E \mid E\otimes \mid E \odot E \mid \\ F &\quad (E,\ldots,E) \mid (E,\ldots,E) \mid [E,\ldots,E] \end{aligned}$$

with variables V, literal constants C, functionals F, prefix operators $\oplus$, postfix operators $\otimes$, and infix operators $\odot$. `( )` denotes a tuple and `[ ]` denotes a list (possibly empty).

3.3. Type Systems

ATMOL is typed because the simple syntax allows too much freedom for the formulation of expressions. For example, it would be syntactically legal to embed a Fortran-like program statement in a numerical expression, while such uses should be prohibited. Instead of imposing constraints at the syntactic level, we choose to use type checking to enforce constraints on the formation of well-formed expressions. In addition, type inference and coercion are used to apply dimensional analysis and grid conversion for model verification and translation.

An ATMOL specification is verified using three different type systems:

Basic types for object types, see Table 2. The types `coordinate` and `index` distinguish objects that can be used as coordinates and objects that can be used to index a grid. Fortran-like programming constructs are of the `statement` type.

Unit types are used in the dimensional analysis of a model. The unit types are internally stored as lists of powers of the basic SI-units. For convenience, units are given by the user as strings that are parsed into the internal SI-unit power lists.

Type	Description	Examples
`array(`τ`, num)`	array with *num* elements of type τ	"`A`" in "`A:i:j`"
`associative(`ρ`->`σ`->`τ`)`	associative operator	"`+`", "`*`", and "`or`"
`boolean`	logical values	"`false`" and "`true`"
`complex`	$\mathbb{C}$	"`complex(1,1/4)`"
`coordinate`	a coordinate variable, a subtype of `float`	"`x`" and "`t`"
`domain(coordinate)`	a coordinate domain	"`x = 0..3/2`"
`domain(index)`	an index domain	"`i = 1..10`"
`float`	$\mathbb{R}$	"`1.2`"
`index`	an index variable, a subtype of `integer`	"`i`"
`integer`	$\mathbb{Z}$	"`7`"
`interval(integer)`	integer interval	"`1..10 step 2`"
`interval(float)`	float interval	"`0.1..5/6 step -0.4`"
`iteration(coordinate)`	a coordinate iteration	"`x = 0..5 step 1/2`"
`iteration(index)`	an index iteration	"`i = 1..10 step k`"
`list(`τ`)`	list of τ	"`[1,2]`"
`lvalue(`τ`)`	assignable τ object, a subtype of τ	"`v`" in "`v:=5`"
`range(integer)`	integer range	"`1..10`"
`range(float)`	float range	"`0.1..5/6`"
`rational`	$\mathbb{Q}$	"`1/2`"
`reference(coordinate)`	a coordinate reference	"`x=0.0`"
`reference(index)`	an index reference	"`i=j+1`"
`statement`	programming statement	"`a:=0`" and "`a:=a+i for i=1..10`"
`string`	string	"`"converged"`"
σ`->`τ	operators and λ-abstractions	"`(a->a+1)`"

Table 2. Basic Types.

Grid types define the type of a grid with respect to a given dimension. Grid types are, for example, staggered finite-difference Arakawa grids [3], finite-volume cells, and spectral grids.

The three type systems are *polymorphic*, support *subtyping*, and allow *parameterized types* and *type variables*. The type systems are separately applied, because the differences in subtyping and the application of type coercions makes it nearly impossible to integrate them into one type system. Type coercions can take place with user-defined type conversion operations. For example, a user might want to automatically convert one type of grid into another by polynomial fitting, which is specified as a conversion operator in the grid type system. The type inference algorithm that implements the three type systems follows a forward/backward scheme [1] and exploits an *iterative deepening* search algorithm [23] to find the "shortest conversion path" for the expression which yields an optimal solution (but one that might not be necessarily unique).

3.4. Aggregate Operators

A novel design choice for ATMOL's syntax and semantics was made with respect to aggregate operations, which are operations performed on the collection of values of a (sub) grid instead of on the value of a single grid point. Such choice of design has an important consequence: all of the typical operators used in scientific models, such as integrations, quadratures, reductions, scans, sums, FFTs, maxval, maxloc, etc., and also certain low-level constructs such as Fortran-like do-loops, are expressed with a uniform notation that involves the use of a local scope of a variable in a construct. The effect of our choice of design is comparable to the effect of variable bindings in λ-expressions.

The syntax has the look-and-feel of Maple's notation of integrals and sums. For example, "`sum(f(i),i=1..10)`" has a local binding of a variable to a range of values. However, the notation used in ATMOL is more fundamental as it adopts this convention for *all* aggregate operators that exhibit local bindings of variables, while Maple has an ad-hoc implementation of bindings that is handled by the code associated with integrals and sums.

More formally, an n-ary functional F, $(n > 1)$, has a *local scope with variable bindings* B for arguments $E_1, \ldots, E_{k-1}$ when the functional is of the form

$$F(E_1, \ldots, E_{k-1}, B, E_k, \ldots, E_{n-1})$$

with expressions E_i, $i = 1, \ldots, n-1$, for some $k = 2, \ldots, n$, where the binding expression B is of the form

$$\begin{aligned} B &\rightarrow B1 \texttt{ by } B \mid B1 \\ B1 &\rightarrow B2 \,\#\, B1 \mid B2 \\ B2 &\rightarrow V = E \mid (B) \end{aligned}$$

Likewise, a dyadic operator $\odot$ of the form $E \odot B$ has a *local scope with variable bindings* B for expression E. The `by` operator can be viewed as a *sequential* construct for composing bindings for multiple variables, while the # operator

Operation	Description
`int(`E`, `B`)`	Integration $\int_B E$
`sum(`E`, `B`)`	Summation $\sum_B E$
`prod(`E`, `B`)`	Product $\prod_B E$
`all(`E`, `B`)`	Logical conjunction of Boolean-typed E over B: $\forall B : E$
`any(`E`, `B`)`	Logical disjunction of Boolean-typed E over B: $\exists B : E$
`maxval(`E`, `B`)`	Maximum value of E over B
`minval(`E`, `B`)`	Minimum value of E over B
`maxloc(`E`, `B`)`	Grid location of maximum value of E over B
`minloc(`E`, `B`)`	Grid location of minimum value of E over B
`loc(`E_1`, `B`, `E_2`)`	Grid location where Boolean-typed E_1 is first true over B, return E_2 if not found
`FFT(`E`, `B`)`	Fourier transform E over B
E `for` B	Loop over program statement E for iterations B
E `forall` B	Parallel do-all loop over program statement E for iterations B
`block(`E_1`, `E_2`, `B`)`	Program statement block E_1 with local variables B (returns value of E_2)
E`@`B	Substitution of variable bindings B in E

Table 3. Operators with Local Bindings.

$$
\begin{array}{rcl|rcl}
FV[\![V]\!] & := & \{V\} \cup V.\text{dependencies} & BV[\![V]\!] & := & \emptyset \\
FV[\![C]\!] & := & \emptyset & BV[\![C]\!] & := & \emptyset \\
FV[\![V := E]\!] & := & FV[\![E]\!] & BV[\![V := E]\!] & := & \{V\} \\
FV[\![E_1\,;\,E_2]\!] & := & (FV[\![E_2]\!] \setminus BV[\![E_1]\!]) \cup FV[\![E_1]\!] & BV[\![E_1\,;\,E_2]\!] & := & BV[\![E_1]\!] \cup BV[\![E_2]\!] \\
FV[\![\oplus E]\!] & := & FV[\![E]\!] & BV[\![\oplus E]\!] & := & BV[\![E]\!] \\
FV[\![E\otimes]\!] & := & FV[\![E]\!] & BV[\![E\otimes]\!] & := & BV[\![E]\!] \\
FV[\![E_1 \odot E_2]\!] & := & FV[\![E_1]\!] \cup FV[\![E_2]\!] & BV[\![E_1 \odot E_2]\!] & := & BV[\![E_1]\!] \cup BV[\![E_2]\!] \\
FV[\![F(E_1,\ldots,E_n)]\!] & := & \bigcup_{i=1}^{n} FV[\![E_i]\!] & BV[\![F(E_1,\ldots,E_n)]\!] & := & \bigcup_{i=1}^{n} BV[\![E_i]\!] \\
FV[\![(E_1,\ldots,E_n)]\!] & := & \bigcup_{i=1}^{n} FV[\![E_i]\!] & BV[\![(E_1,\ldots,E_n)]\!] & := & \bigcup_{i=1}^{n} BV[\![E_i]\!] \\
FV[\![[E_1,\ldots,E_n]]\!] & := & \bigcup_{i=1}^{n} FV[\![E_i]\!] & BV[\![[E_1,\ldots,E_n]]\!] & := & \bigcup_{i=1}^{n} BV[\![E_i]\!] \\
FV[\![E \odot B]\!] & := & (FV[\![E]\!] \setminus BV[\![B]\!]) \cup FV[\![B]\!] & BV[\![E \odot B]\!] & := & BV[\![E]\!] \cup BV[\![B]\!]
\end{array}
$$

$$
FV[\![F(E_1,\ldots,E_{k-1},B,E_k,\ldots,E_n)]\!] := (\bigcup_{i=1}^{k-1} FV[\![E_i]\!] \setminus BV[\![B]\!]) \cup \bigcup_{i=k}^{n} FV[\![E_i]\!] \cup FV[\![B]\!]
$$

$$
BV[\![F(E_1,\ldots,E_{k-1},B,E_k,\ldots,E_n)]\!] := \bigcup_{i=1}^{k-1} FV[\![E_i]\!] \cup \bigcup_{i=k}^{n} BV[\![E_i]\!] \cup BV[\![B]\!]
$$

$$
\begin{array}{rcl|rcl}
FV[\![B_1 \texttt{ by } B_2]\!] & := & (FV[\![B_1]\!] \setminus BV[\![B_2]\!]) \cup FV[\![B_2]\!] & BV[\![B_1 \texttt{ by } B_2]\!] & := & BV[\![B_1]\!] \cup BV[\![B_2]\!] \\
FV[\![B_1 \# B_2]\!] & := & FV[\![B_1]\!] \cup FV[\![B_2]\!] & BV[\![B_1 \# B_2]\!] & := & BV[\![B_1]\!] \cup BV[\![B_2]\!] \\
FV[\![V = E]\!] & := & \text{FV[\![E]\!]} & BV[\![V = E]\!] & := & \{V\}
\end{array}
$$

Fig. 7. Free and Bound Variables.

serves as a cross operation or *parallel* construct for binding variables. The scope of bindings is limited to the arguments at the left of the binding expression B. See Table 3 for examples.

3.5. Free and Bound Variables

The notion of a scope of bindings is formalized by the sets of free and bound variables of an expression E shown in Figure 7.

In the figure, "V.dependencies" denotes the set of variables on which V depends as declared by the *coord* part of the declaration of V. For example, the variable `u` declared in

```
u :: float dim "m/s"
  field (x(half),y(grid),z(grid))
     on i=1..n by j=1..m by k=1..l.
```

has $u.\text{dependencies} = \{x, y, z\}$. As a result, expressions with aggregate operations such as $\int_0^1 u\,dx$ now make sense, because variable u depends on variable x.

ATMOL's substitution algorithm replaces free variables in expressions with new values. A substitution is specified with the @-operator, see Table 3. For example, "E`@(i=i+1)`" replaces all free occurrences of `i` in E by `i+1`. This substitution algorithm may rename bound variables in expressions to avoid name clashes, which is comparable to α-conversion as applied by β-reduction in λ-calculus. The substitution algorithm is frequently used for partial evaluation and in the simplification of expressions.

3.6. Array Variables

The sets of free and bound variables provide a means to check the dimensionality of multi-dimensional objects returned as a result of an expression. The numerical codes of scientific models frequently make use of arrays for storage. Synthesis of numerical codes requires introduction of (temporary) arrays with proper element types, dimensionality, and accurate array bounds. This is a non-trivial problem for the implementation of numerical schemes in program code. The concept of local bindings and free variables can be used to accurately establish the dimensionality of (temporary) arrays and their array bounds.

Array Index Analysis The set of free index variables of an expression E, denoted $IV[\![E]\!]$, is defined by

$$IV[\![E]\!] := FV[\![E]\!] \cap \{V | V \text{ is an index variable}\}$$

This set describes the index space in which the expression is evaluated, which typically constitutes the grid space of the PDE problem. To generate code for E for numerical evaluation, a loop is constructed that assigns the value of E to an array variable, say `u`:

`u:i:j:k :=` E `forall i=1..n by j=1..m by k=1..l`

where we assumed that $FV[\![E]\!] = \{\texttt{i}, \texttt{j}, \texttt{k}\}$.

Array Bound Analysis Consider for example the free index variables of the symbolic expression $\sum_{k=1}^{l} A_{i,k}B_{k,j}$ which is the set $IV[\![\texttt{sum(A:i:k*B:k:j, k=1..l)}]\!] = \{\texttt{i}, \texttt{j}\}$ Hence, the result can be stored in a two dimensional array, e.g. `C` as in the following program fragment

```
C:i:j := sum(A:i:k*B:k:j, k=1..l)
   forall i=1..n by j=1..m
```

which computes the matrix product of A and B and assigns the result to C. A *domain inference* and *value range propagation* [6] algorithm based on the substitution algorithm derives the array bounds of array variables in the target code. In the example code fragment, the array bounds `A:(1..n):(1..l)`, `B:(1..l):(1..m)`, and `C:(1..n):(1..m)` are derived.

The initial grid domains declared for the fields of a model may be extended by the domain inference to ensure that in the generated codes no references will be made to grid points outside of the domain.

3.7. Dynamic Semantics, Rewrites, and Partial Evaluation

A classification mechanism in CTADEL allows for defining a hierarchy of objects and functionals with algebraic properties. Figure 8 depicts the predefined hierarchy of operator classes. This framework allows for the automatic simplification of an operator belonging to a class of objects without the need for specifying individual rewrites for this operator.

Example instances of `reduction_op` are `int`, `sum`, `prod`, `all`, `any`, `maxval`, and `minval`. An operator of the `differentiation_op` class inherits properties of the `self_commuting_op`, `commuting_op`, `operator`, and `linear_op` classes. Examples are the partial derivative $\mathtt{d}E/\mathtt{d}V$ operator and all finite difference operators. An example integration operator is `int` and also quadratures are considered integration operators.

An abstract operator `abstract_op` is an overloaded operator with no implementation. Instead, the operator will be replaced by a concrete operator from a list of `choices` that matches the types used in the context of the operator. Examples are the "`df`" and "`int`" operators.

CTADEL's term rewriting system (TRS) is implemented modulo associativity and commutativity (AC), i.e. associativity and commutativity of operators are implicitly exploited in matching rewrite rules. CTADEL's TRS is also implemented modulo operator commuting from operator commuting diagrams. Many operators in scientific models are linear and exhibit commuting properties, such as integrals, quadratures, derivatives, finite differences, interpolations, sums, FFTs, etc. The commuting relationships between these operators is often graphically described using operator commutativity diagrams. These commuting properties can be declared in ATMOL and are implicitly used by CTADEL for pattern matching in the application of rewrite rules. For example, when functionals F' and F'' are declared to commute, then

$$\begin{aligned} &F'(F''(E, E''_1, \ldots, B'', \ldots, E''_m), \\ &\quad E'_1, \ldots, B', \ldots, E'_n) \\ &= F''(F'(E, E'_1, \ldots, B', \ldots, E'_n), \\ &\quad E''_1, \ldots, B'', \ldots, E''_m) \end{aligned} \tag{4}$$

if $(\bigcup_{i=1}^{m} FV[\![E''_i]\!] \cup FV[\![B'']\!]) \cap BV[\![B']\!] = \emptyset$ and $(\bigcup_{i=1}^{n} FV[\![E'_i]\!] \cup FV[\![B']\!]) \cap BV[\![B'']\!] = \emptyset$ and $BV[\![B']\!] \cap BV[\![B'']\!] = \emptyset$. This constraint is

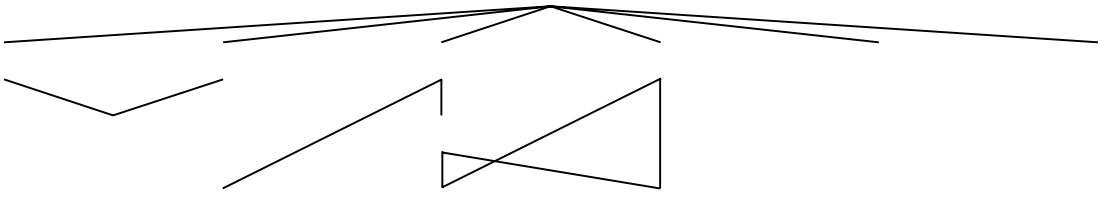

Fig. 8. Operator Class Hierarchy.

imposed to prevent variable name clashes after the interchange. For example, the rule to simplify aggregate linear operators of the `linear_op` class is

$$F(E_1 * E_2, E_3, \ldots, B, \ldots, E_n) \Rightarrow E_1 * F(E_2, E_3, \ldots, B, \ldots, E_n) \quad (5)$$

if F is a `linear_op` and $FV[\![E_1]\!] \cap BV[\![B]\!] = \emptyset$. Consider for example the application of rule (5):

$$\texttt{sum}(\texttt{sum}(\texttt{f}(\texttt{i},\texttt{j}) * \texttt{i}, \texttt{i} = 1..\texttt{n}), \texttt{j} = 1..\texttt{m}) \Rightarrow \texttt{sum}(\texttt{i} * \texttt{sum}(\texttt{f}(\texttt{i},\texttt{j}), \texttt{j} = 1..\texttt{m}), \texttt{i} = 1..\texttt{n})$$

The commuting properties of sums are exploited to interchange the sums when the rule is applied.

3.8. Code Synthesis with Templates

All higher-level functional operators of ATMOL are translated to procedural program code constructs in ATMOL that resemble Fortran programming statements. This intermediate program representation is output in Fortran by a pretty printer. The translation to intermediate code takes place through the application of template definitions for operators. For example, the template definition of the higher-order "`reduce`" operator in ATMOL is:

```
reduce(E :: _T, B :: domain(index),
Op :: associative(_T->_T->_T)) :: _T function
  { reduce := unit_element(Op);
    reduce := apply_op(Op, reduce, E) for B
  }.
```

Templates are type checked to ensure that the code they contain is statically correct. They are not checked for dimensional units or grid compatibility, as the templates are applied at a later stage of the translation after discretization through grid inference and coercion. The example template above uses a type variable `_T` to define the type of the expression, the type of the associative operator, and the type of the result of the operator. This essentially defines the following rewrite rule:

$$\texttt{reduce}(E, B, \odot) \Rightarrow \texttt{block}(V{:=}U\,;\ V{:=}V \odot E \texttt{ for } B,\ V,\ V = E.\text{type}) \quad (6)$$

with a new variable V such that $V \notin FV[\![E]\!]$, where U is the unit element of the group with operator $\odot$. Note that the set of free variables is not changed by the rewrite. The "`block`" construct forms a "codelet". When templates are applied the codelets are combined and optimized to form a sequence of program statements. An example of code generation with templates and optimization of codelets is shown in Figure 9.

The process shown in Figure 9 is lengthy and requires "smart" rules to discover that the double summation can be performed in one loop. The recognition of opportunities for optimization at the low-level program code is limited as the code gets more complicated and side-effects get in the way of an accurate analysis. In most cases however, optimizations can be performed at a higher level much more easily and this can have a significant impact on the quality and efficiency of the synthesized codes. For example, consider the rewrite

$$\texttt{reduce}(\texttt{reduce}(E,B_1,\odot),B_2,\odot) \Rightarrow \texttt{reduce}(E,B_1 \texttt{ by } B_2,\odot) \quad (7)$$

$$\begin{aligned}
&\texttt{reduce}(\texttt{reduce}(\texttt{f}(\texttt{i}), \texttt{i} = 1..\texttt{j}, +), \texttt{j} = 1..\texttt{n}, +)\\
&\quad\Rightarrow\ \texttt{reduce}(\texttt{block}(\texttt{S} := 0; \texttt{S} := \texttt{S} + \texttt{f}(\texttt{i}) \texttt{ for } \texttt{i} = 1..\texttt{j}, \texttt{S}, \texttt{S} = \texttt{integer}), \texttt{j} = 1..\texttt{n}, +)\\
&\quad\Rightarrow\ \texttt{block}(\texttt{S} := 0; \texttt{block}(\texttt{S}' := 0; \texttt{S}' := \texttt{S}' + \texttt{f}(\texttt{i}) \texttt{ for } \texttt{i} = 1..\texttt{j}, \texttt{S} := \texttt{S} + \texttt{S}', \texttt{S}' = \texttt{integer}) \texttt{ for } \texttt{j} = 1..\texttt{n}, \texttt{S}, \texttt{S} = \texttt{integer})\\
&\quad\Rightarrow\ \texttt{block}(\texttt{S} := 0; ((\texttt{S}' := 0; \texttt{S}' := \texttt{S}' + \texttt{f}(\texttt{i}) \texttt{ for } \texttt{i} = 1..\texttt{j}); \texttt{S} := \texttt{S} + \texttt{S}') \texttt{ for } \texttt{j} = 1..\texttt{n}, \texttt{S}, \texttt{S} = \texttt{integer by } \texttt{S}' = \texttt{integer})\\
&\quad\Rightarrow\ \texttt{block}(\texttt{S} := 0; \texttt{S} := \texttt{S} + \texttt{f}(\texttt{i}) \texttt{ for } \texttt{i} = 1..\texttt{j} \texttt{ for } \texttt{j} = 1..\texttt{n}, \texttt{S}, \texttt{S} = \texttt{integer})
\end{aligned}$$

```
      INTEGER S
      S=0
      DO 10 j=1,n
      DO 10 i=1,j
      S=S+f(i)
10    CONTINUE
```

Fig. 9. Example Template Expansion, Codelet Optimization, and Fortran Output.

which translates multiple reductions into a single reduction with combined domains and could be applied to the first line of Figure 9. It is easy to verify that this rewrite does not change the set of free variables:

$$FV[\![\texttt{reduce}(\texttt{reduce}(E,B_1,\odot),B_2,\odot)]\!] = (FV[\![E]\!] \setminus BV[\![B_1]\!]) \setminus BV[\![B_2]\!]$$

and

$$FV[\![\texttt{reduce}(E,B_1\,\texttt{by}\,B_2,\odot)]\!] = FV[\![E]\!] \setminus (BV[\![B_1]\!] \cup BV[\![B_2]\!])$$

which are identical sets. This rule enables a mapping to code in one step using rule 6:

```
reduce(f(i), i = 1..j by j = 1..n, +) ⇒
      block(S := 0;S := S + f(i)
for i = 1..j by j = 1..n, S, S = integer)
```

This example illustrates the importance of high-level optimizations to avoid potential difficulties with low-level optimizations. The TRS formed by the translation rules is not confluent which is intentional as its purpose is to enable code optimizations instead of code normalization which are often contradictory goals.

4. Conclusions

In this paper, we highlighted the features of ATMOL, discussed its design and implementation, and demonstrated the importance of its use in the code generation process of atmospheric models. To our knowledge, the ATMOL language and the code synthesis approach are novel and is a first attempt to integrate high-level optimizations with low-level optimizations in a unified language and framework for problem refinement and code synthesis.

Acknowledgements

I would like to thank Gerard Cats of the Royal Netherlands Meteorological Institute for his advise. I am grateful to Lex Wolters for his support for the development of CTADEL.

References

[1] A. AHO, R. SETHI, AND J. ULLMAN. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading MA, 1985.

[2] R. L. AKERS, E. KANT, C. J. RANDALL, S. STEINBERG, AND R. L. YOUNG. SciNapse: A problem-solving environment for partial differential equations. *IEEE Computational Science & Engineering*, 4(3):32–42, July/September 1997.

[3] A. ARAKAWA AND V. R. LAMB. A potential enstrophy and energy conserving scheme for the shallow water equations. *Monthly Weather Review*, 109:18–36, 1981.

[4] E. ARGE, A. M. BRUASET, AND H. P. LANGTANGEN, EDITORS. *Modern Software Tools for Scientific Computing*. Birkäuser, 1997.

[5] L. BATH, J. OLSON, AND J. ROSINSKI. *User's Guide to NCAR CCM2*. National Center for Atmospheric Research, Boulder, Colorado, 1992.

[6] W. BLUME AND R. EIGENMANN. Demand-driven, symbolic range propagation. In 8th *International workshop on Languages and Compilers for Parallel Computing*, pages 141–160, Columbus, Ohio, USA, August 1995.

[7] G. CATS AND L. WOLTERS. The Hirlam project. *IEEE Computational Science & Engineering*, 3(4):4–7, 1996.

[8] D. DENT, L. ISAKSEN, G. MOZDZYNSKI, M. O'KEEFE, G. ROBINSON, AND F. WOLLENWEBER. IFS model: Performance measurements. In 6th *ECMWF Workshop on the use of Parallel Processors in Meteorology*, Reading, UK, November 1994. ECMWF.

[9] E. KÅLLÉN (ED.). *HIRLAM Documentation Manual System 2.5*. Norrköping, Sweden, June 1996. Available from SMHI, S–60176.

[10] E. GALLOPOULOS, E. HOUSTIS, AND J. R. RICE. Computer as thinker/doer: Problem-solving environments for computational science. *IEEE Computational Science &Engineering*, 1:11–23, 1994.

[11] D. GANNON, R. BRAMLEY, T. STUCKEY, J. VILLACIS, J. BALASUBRAMANIAN, E. AKMAN, F. BREG, S. DIWAN, AND M. GOVINDARAJU. Developing component architectures for distributed scientific problem solving. *IEEE Computational Science &Engineering*, 1998.

[12] JR. G. O. COOK AND J. F. PAINTER. ALPAL: A tool to generate simulation codes from natural descriptions. *Expert Systems for Scientific Computing*, 1992.

[13] W. GROPP, R. LUSK, AND A. SKJELLUM. *Using MPI*, 1994.

[14] G. J. HALTINER AND R. T. WILLIAMS. *Numerical Prediction and Dynamic Meteorology*. John Wiley &Sons, New York, second edition, 1980.

[15] W. HIBBARD AND D. SANTEK. The VIS-5D system for easy interactive visualization. In *IEEE Visualization '90*, pages 129–134, 1990.

[16] E. N. HOUSTIS, E. GALLOPOULOS, R. BRAMLEY, AND J. R. RICE. Problem-solving environments for computational science. *IEEE Computational Science &Engineering*, 4(3):18–21, July/September 1997.

[17] E. N. HOUSTIS, J. R. RICE, N. P. CHRISOCHOIDES, H. C. KARATHANASIS, P. N. PAPACHIOU, M. K. SAMARTZIS, E. A. VAVALIS, KO-YANG WANG, AND S. WEERAWARANA. //ELLPACK: A numerical simulation programming environment for parallel MIMD machines. In 4th *ACM International Conference on Supercomputing*, pages 96–107, New York, 1990. ACM Press.

[18] The Math Works Inc. *MATLAB, High-Performance Numeric Computation and Visualization Software, User's Guide*, 1992.

[19] J. L. MCGREGOR, H. L. GORDON, I. B. WATTERSON, AND M. R. DIX. The CSIRO 9-level atmospheric general circulation model. Technical paper 26, CSIRO Division of Atmospheric Research, Mordialloc, VIC, 1992.

[20] M. F. P. O'BOYLE AND J. M. BULL. Expert programmer versus parallelizing compiler: A comparative study of two approaches for distributed shared memory. *Scientific Programming*, 5:63–88, 1996.

[21] C. PANCAKE AND D. BERGMARK. Do parallel languages respond to the needs of scientific programmers? *IEEE Computing*, 23(12):13–23, December 1990.

[22] J. R. RICE AND R. F. BOISVERT. *Solving Elliptic Problems Using ELLPACK*. Springer Verlag, New York, 1985.

[23] E. RICH AND K. KNIGHT. *Artificial Intelligence*. McGraw Hill, New York, 2nd ed edition, 1991.

[24] CLEMENS SZYPERSKI. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[25] P. TU AND D. PADUA. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In 9th *ACM International Conference on Supercomputing*, pages 414–423, New York, July 1995. ACM Press.

[26] Y. UMETANI, M. TSUJI, K. IWASAWA, AND H. HIRAYAMA. DEQSOL: A numerical simulation language for vector/parallel processors. In B. Ford and F. Chatelin, editors, *Problem Solving Environments for Scientific Computing*, Amsterdam, 1987. North-Holland.

[27] R. A. VAN ENGELEN. CTADEL: *A Generator of Efficient Numerical Codes*. PhD thesis, Leiden University, The Netherlands, 1998. ISBN 90-12014-9.

[28] R. A. VAN ENGELEN, I. HEITLAGER, L. WOLTERS, AND G. CATS. Incorporating application dependent information in an automatic code generating environment. In 11th *ACM International Conference on Supercomputing*, pages 180–187, New York, 1997. ACM Press.

[29] R. A. VAN ENGELEN, L. WOLTERS, AND G. CATS. CTADEL: A generator of multi-platform high performance codes for PDE-based scientific applications. In 10th *ACM International Conference on Supercomputing*, pages 86–93, New York, 1996. ACM Press.

[30] R. A. VAN ENGELEN, L. WOLTERS, AND G. CATS. The CTADEL application driver for numerical weather forecast systems. In A. Sydow, editor, 15th *IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics (session: Problem Solving Environments for Scientific Computing)*, volume 4, pages 571–576, Berlin, Germany, 1997. Wissenshaft &Technik Verlag.

[31] R. A. VAN ENGELEN, L. WOLTERS, AND G. CATS. Tomorrow's weather forecast: Automatic code generation for atmospheric modeling. *IEEE Computational Science &Engineering*, 4(3):22–31, July/September 1997.

Received: July, 2001
Revised: October, 2001
Accepted: November, 2001

Contact address:
Robert A. van Engelen
Department of Computer Science
Florida State University
Tallahassee
FL 32306-4530
USA
e-mail: engelen@cs.fsu.edu

ROBERT VAN ENGELEN received his M. Sc. (1994) in Computer Science from the University of Utrecht, the Netherlands, and his Ph. D. (1998) in Computer Science from the University of Leiden, the Netherlands. He is currently Assistant Professor in the Department of Computer Science at Florida State University. His research interests include problem-solving environments for scientific computation, compiler analysis and optimization, high-performance computing, and probabilistic networks. He received two National Science Foundation grants and has published over 20 papers in refereed conferences and journals such as IEEE Transactions on Pattern Analysis and Machine Intelligence, Real-Time Systems Journal, and IEEE Journal of Computational Science and Engineering. He is a member of IEEE, ACM, and IMACS.