Scientific Programming 14 (2006) 27–40
IOS Press

Formal constraints on memory management for composite overloaded operations

Damian W.I. Rouson[a], Xiaofeng Xu[b] and Karla Morris[c]
[a]*US Naval Research Laboratory, 4555 Overlook Ave. SW, Washington, DC 20375, USA*
Tel.: +1 202 767 6965; Fax: +1 815 572 8203; E-mail: damian.rouson@nrl.navy.mil
[b]*Department of Fire Protection Engineering, University of Maryland, College Park, MD 20742, USA*
Tel.: +1 301 405 3990; Fax: +1 301 405 9383; E-mail: xxf@umd.edu
[c]*Mechanical Engineering Dept., The Graduate Center of The City University of New York, Fifth Ave. at 34th St. New York, NY, USA*
Tel.: +1 212 650 7134; Fax: +1 212 650 8314

Abstract The memory management rules for abstract data type calculus presented by Rouson, Morris & Xu [15] are recast as formal statements in the Object Constraint Language (OCL) and applied to the design of a thermal energy equation solver. One set of constraints eliminates memory leaks observed in composite overloaded expressions with three current Fortran 95/2003 compilers. A second set of constraints ensures economical memory recycling. The constraints are preconditions, postconditions and invariants on overloaded operators and the objects they receive and return. It is demonstrated that systematic run-time assertion checking inspired by the formal constraints facilitated the pinpointing of an exceptionally hard-to-reproduce compiler bug. It is further demonstrated that the interplay between OCL's modeling capabilities and Fortran's programming capabilities led to a conceptual breakthrough that greatly improved the readability of our code by facilitating operator overloading. The advantages and disadvantages of our memory management rules are discussed in light of other published solutions [11,19]. Finally, it is demonstrated that the run-time assertion checking has a negligible impact on performance.

1. Introduction

Formal methods form an important branch of software engineering that has apparently been applied to the design of only a small percentage of scientific simulation programs [3,8]. Two pillars of formalization are specification and verification – that is specifying mathematically what a program must do and verifying the correctness of an algorithm with respect to the specification. The numerical aspects of scientific programming are already formal. The mathematical equations one wishes to solve in a given scientific simulation provide a formal specification, while a proof of numerical convergence provides a formal verification. Hence, formal methods developers often cite a motivation of seeking correctness standards for non-scientific codes as rigorous as those for scientific codes [13]. This ignores, however, the non-numerical aspects of scientific programs that could benefit from greater rigor. One such aspect is memory management. The current paper specifies formal constraints on memory allocations in a Fortran 95/2003 program for simulating thermal conduction and convection.

There have been longstanding calls for increased use of formal methods in scientific programming to improve reliability. Nearly a decade ago, Stevenson proclaimed a "crying need for highly reliable system simulation methodology, far beyond what we have now," He cited formal methods amongst the strategies that might improve the situation [18]. Part of his basis was Hatton's measurement of an average of 10 statically detectable serious faults

ISSN 1058-9244/06/$17.00

per 1,000 lines amongst millions of lines of commercially released scientific C and Fortran 77 code [9]. Hatton defined a fault as "a misuse of the language which will very likely fail in some context." Examples include interface inconsistencies and using un-initialized variables. Hatton suggested that formal methods reduce defect density by a factor of three (cited in [18]).

Pace [14] expressed considerable pessimism about the prospects for the adoption of formal methods in scientific simulation because of the requisite mathematical training, which often includes set theory and a predicate calculus. A good candidate for adoption must balance such rigor with ease of use. The Object Constraint Language (OCL) strikes such a balance by facilitating the expression of formal statements about software models without the use of mathematical symbols known only to formal methods specialists. A primary requirement stated by OCL's designers is that OCL "must be understood by people who are not mathematicians or computer scientists" [21].

To attract scientific programmers, any software development strategy must address performance. Fortunately, program specification and verification are part of the software design rather than the implementation. Thus, they need not impact run-time performance. However, run-time checking of assertions, a third pillar of formal methods, *is* part of the implementation [5]. Nonetheless, when the assertions are simple Boolean expressions, they often occupy a negligible fraction of the run time compared to the long loops over millions of floating point calculations typical of scientific software.

A final factor influencing adoption of formal methods is the lack of a common approach for describing the structure of traditional scientific codes beyond flow charts. OCL's incorporation into the recent versions of the Unified Modeling Language (UML) [21], a graphical standard for describing software structure and behavior, suggests that newcomers must simultaneously leap two hurdles: learning OCL and learning UML. Fortunately, increasing interest in object-oriented scientific programming has led to more frequent scientific program structural descriptions resembling UML class models [1,2]. Class models describe object interfaces and relationships between classes. Object interfaces describe object behavior (procedures) and state (data).

The coupling of OCL and UML class models represents a subtle yet important shift away from the traditional emphases of scientific programming. Traditional approaches develop mathematical abstractions for the physics and the numerics but not the software. For example, continuum mechanics is an abstraction of condensed matter in that it retains only the level of detail required to model macroscopic phenomena. Numerical approximations represent an abstraction of the governing continuum equations in that they retain only the number of discrete values required to obtain a solution within a given tolerance. Likewise, UML class models are software abstractions that retain only the features needed to describe object interfaces and their interrelationships. OCL facilitates describing the resulting software model formally.

The goals of this paper are twofold:

1. to demonstrate how exposure to formal methods benefited us greatly by inspiring a systematic approach to run-time assertion checking and
2. to demonstrate an increase in code readability achieved because OCL forced us to think abstractly about an otherwise language-specific construct: pointers.

Before returning to these themes in Sections 3 and 4 of this paper, Section 2 presents a UML class model for simulating thermal conduction and convection. Section 2 also presents a Fortran 95/2003 implementation. Section 3 specifies a set of OCL constraints intended to prevent memory leaks and reduce the code's dynamic memory requirements. Section 4 discusses limitations of our approach and describes our run-time assertions and performance results. Section 5 summarizes our conclusions.

2. Case study: a thermal conduction and convection solver

2.1. Problem statement

OCL constraints are meaningful only in the context of an object-oriented class model [21]. Figure 1 presents a UML class model for a scalar advection/diffusion solver. The random molecular diffusion and organized transport (advection) of scalar quantities is of broad scientific interest. Applications range from modeling combustion to

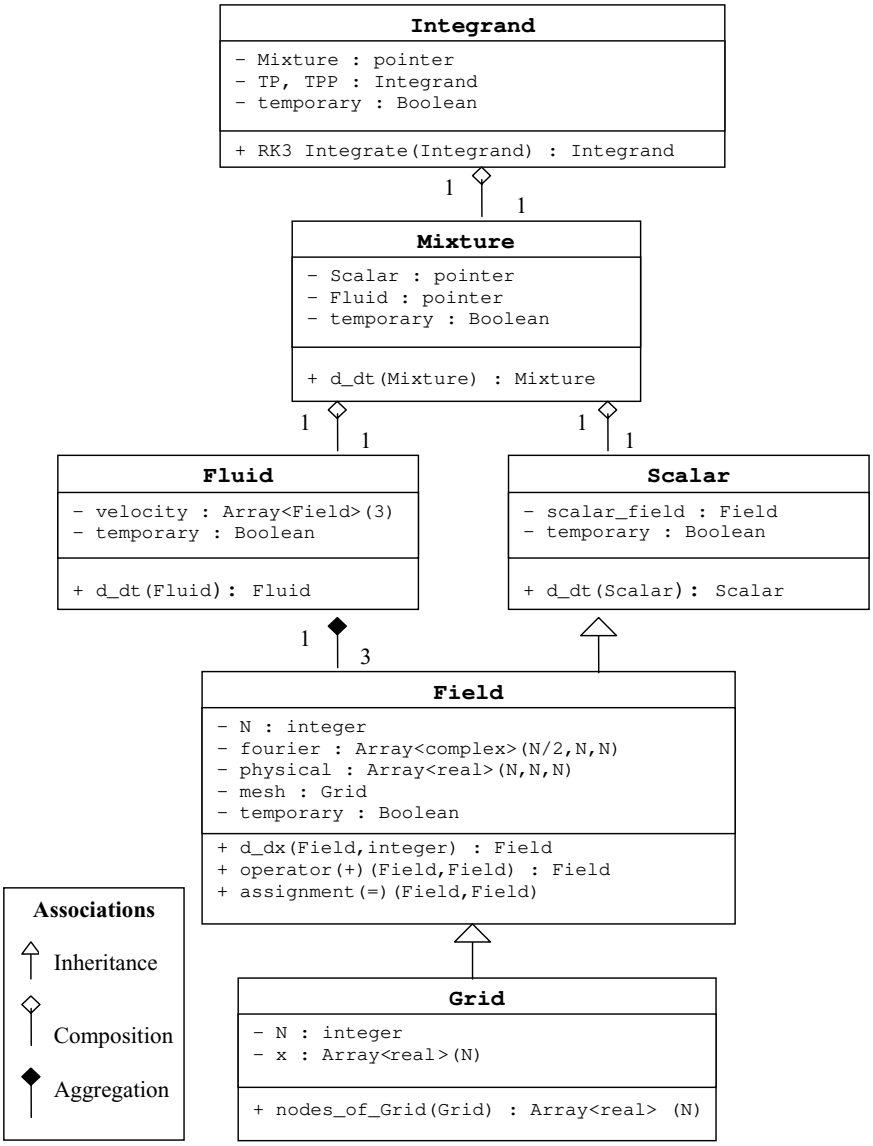
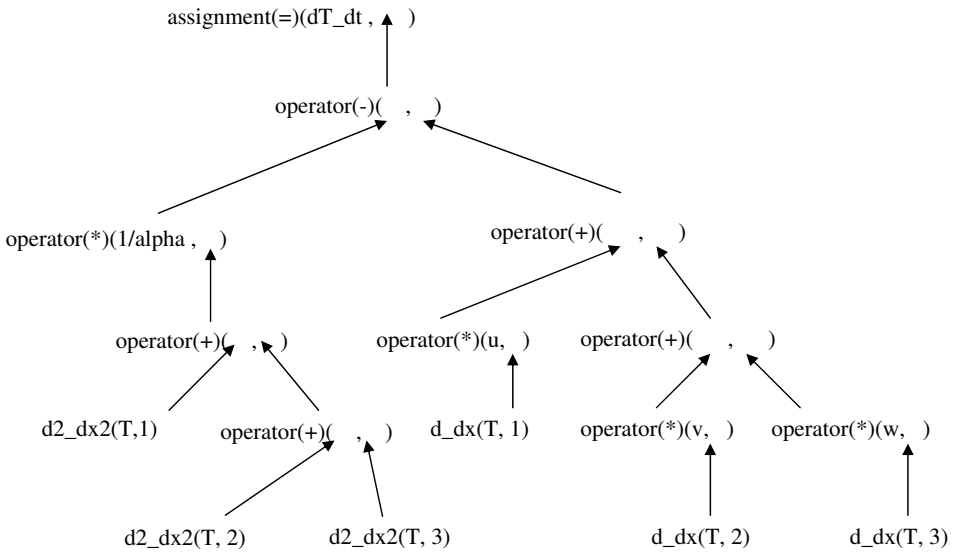


Fig. 1. Navier-Stokes fluid and scalar advection/diffusion class model (numbers near associations indicate the number of instantions present on either side of the association).

tracking atmospheric pollutants. This paper examines a case where the scalar is temperature and the resulting scalar advection and diffusion are thermal convection and conduction, respectively.

Figure 1 depicts three types of relationships between classes: inheritance, composition and aggregation. As defined by Decyk, Norton and Szymanksi [6,7], inheritance relationships are one-to-one associations referred to as "is a" relationships. Composition represents a "contains a" relationship wherein the contained object survives if the containing object ceases to exist. Aggregation represents a "has a" relationship wherein each instance of a class includes one or more instances of another class that does not survive if the including object ceases to exist. Thus, Fig. 1 reads as follows: an Integrand contains a Mixture that contains a Fluid and a Scalar; whereas a Fluid has three Fields (velocity components), a Scalar is a Field, and a Field is a Grid with an associated set of values. The Integrand uses Mixture time derivatives to integrate the Mixture forward in time. The Mixture time derivative is the vector containing its component Scalar and Fluid time derivatives. The Fluid time derivative is calculated from the Navier-Stokes equations as described elsewhere [17]. The Scalar time derivative is calculated from the thermal




```
dT_dt = (1/alpha)*( d2_dx2(T,1) + d2_dx2(T,2) + d2_dx2(T,3) ) &
                  -( u*d_dx(T,1))+ v*d_dx(T,2))+ w*d_dx(T,3) )
```

Fig. 2. Source code (below) and call tree (above) resulting from evaluating the right-hand side of the equation for thermal conduction and convection (Eq. (1)).

conduction/convection equation described below. The Grid holds mesh positions and connectivity. The Mixture mediates information exchange between the Scalar and Fluid to avoid the circular reference that would result from making the Scalar and Fluid aware of each other.

A scalar temperature field, $T(x, y, z, t)$, in a homogeneous fluid medium satisfies the following advection/diffusion relation, also referred to as the "energy equation":

$$\frac{\partial T}{\partial t} = \frac{1}{\alpha}\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) - \left(u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y} + w\frac{\partial T}{\partial z}\right) \tag{1}$$

where $\mathbf{u} = (u, v, w)^T$ is the Fluid velocity field and α is the thermal diffusivity. The first parenthetical terms on the right-hand side (RHS) of Eq. (1) model thermal diffusion, which is energy transport by random molecular motions. The second parenthetical terms model thermal advection, or "convection", which is energy transport by organized motions (fluid flow). The operators required to evaluate the RHS of Eq. (1) include the arithmetic operators $\times$, $+$, and $-$, the differential operators $\partial/\partial x$, $\partial/\partial y$, and $\partial/\partial z$ and combinations thereof. Figure 2 shows the Fortran 95/2003 source code and resulting call tree for evaluating the RHS of Eq. (1).

We recently published *informal* memory management rules for using the Integrand class to integrate the time derivative on the left-hand side of evolutions equations such as Eq. (1) [15]. (A specification in any natural language such as English is inherently informal due to the ambiguities associated with such languages.) In the current paper, we turn our attention to evaluating the RHS of Eq. (1). The size of the 3D grids on which the spatial derivatives of T are needed implies the state vectors occupy much more memory than did the state vectors considered in our previous papers [15,16]. Those papers treated very different objects: one-dimensional quantum vortices and zero-dimensional droplets (point masses). The increased memory requirements of 3D fields motivate the current paper.

Each procedure call in Fig. 2 produces a Field result that is no longer needed once the operator of lower precedence (higher on the call tree) completes. Thus, Fig. 2 represents 15 intermediate instantiations. Each intermediate result is an instance of the Scalar class containing its own Field representation. Simulating turbulent flows at laboratory conditions requires discrete fields with roughly 512^3 grid points [4]. With 4-byte precision per value, each Field instance occupies 0.5 gigabytes. If compilers do not deallocate the intermediate results, the burden falls on the

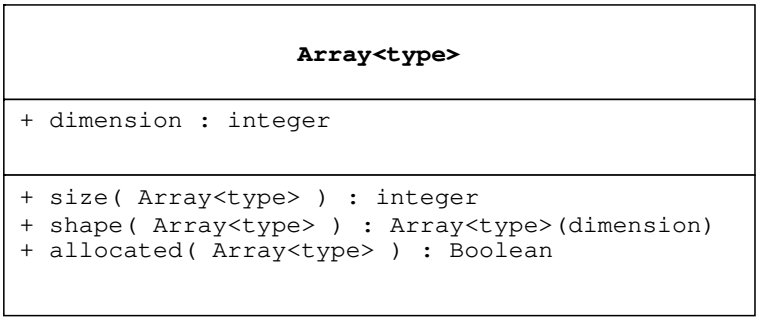


Fig. 3. Sample utility array class template.

programmer to do so. When the final assignment at the top of the call tree is executed, the names associated with intermediate allocations are out of scope and it would be difficult or impossible for the programmer to free the associated memory at that point. Rouson, Morris and Xu (hereafter "RMX") [15] developed the following definitions and rules for freeing the memory allocated by each overloaded operator in Fortran 95/2003:

Definition 1. Given an object that appears as the result of an arithmetic or differential operator, we define the object as *temporary* if it can be deleted at the termination of execution of the first subsequent operator in which it appears as an argument.

Corollary: All objects that are not temporary are *persistent*.

RMX implemented this scheme by inserting Boolean (Fortran LOGICAL) flags named temporary in the definition of each data structure. They then applied four rules to memory allocations and de-allocations:

Rule 1: All arithmetic and differential operator results are marked as temporary upon creation.
Rule 2: Left-hand arguments to defined assignments are marked as persistent.
Rule 3: Temporary objects are deleted prior to the termination of any arithmetic or differential operator in which they appear as an argument.
Rule 4: Persistent objects are deleted prior to the termination of the procedure that instantiated them.

Before explaining in Sections 3–4 how formalizing these rules greatly benefited us, the next two subsections address specification issues in OCL and implementation issues in Fortran.

2.2. Modeling arrays with OCL

Multidimensional arrays account for the lion's share of the memory allocated in our Field implementation. Although OCL has a "sequence" type that could naturally be used to model one-dimensional arrays, OCL does not contain an intrinsic multidimensional array type. Any types defined by a UML model, however, are considered OCL model types [15]. We will assume the user has access to a multidimensional utility array class template for constructing collections of various basic types and model types. Figure 3 depicts such a template. We will not explore its implementation details here. The interested reader might consult the text by Barton and Nackman [2] for an example of a C++ array template class written for scientific and engineering applications. For our purposes, it will suffice to assume that the available array class implements the attributes and services of the Fortran 95/2003 arrays. In particular, the class should have public methods that return the array size, shape, and allocation status as shown in Fig. 3.

2.3. Fortran 95/2003 implementation

Although OCL is programming language-independent, it will prove fruitful to discuss our programming language to facilitate the discussions in Section 3. Two important reasons for choosing Fortran 95/2003 are the language's operator overloading and complex number type. Section 4 demonstrates that a complex Fast Fourier Transform (FFT) occupies the largest share of our code's execution time. We refer the reader to Rouson and Xiong [16] for a

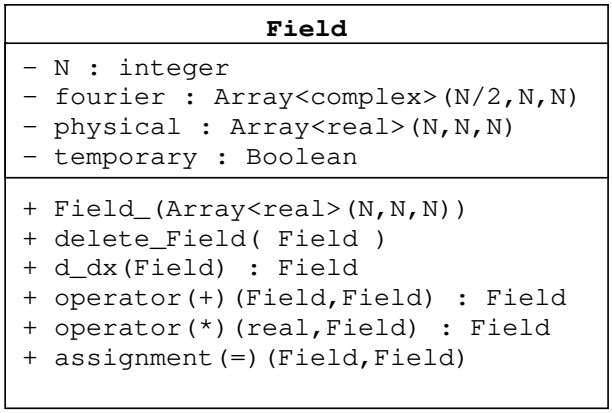


Fig. 4. Field class arithmetic and differential operators.

more complete discussion of our language choice. We also cite Akin [1] for his clear exposition of object-oriented programming (OOP) techniques in Fortran 90/95.

To provide the requisite operators, we must define overloaded operators of the form

```
PRIVATE ! hide all data & procedures by default
PUBLIC ::  OPERATOR (+), Field

INTERFACE OPERATOR (+)
  MODULE PROCEDURE field plus field
END INTERFACE OPERATOR (+)
```

where the exclamation marks precede comment. The above syntax tells the complier to call `field plus field()` whenever it finds the "+" operator between two objects that match the argument list in the following function signature:

```
FUNCTION field plus field (left, right) RESULT (total)
TYPE (Field), INTENT (IN) ::  left, right
TYPE (Field) ::  total
```

The above function returns the pointwise sum of two instances of the following derived type:

```
TYPE Field
  PRIVATE
    REAL , DIMENSION (:,:,:), ALLOCATABLE ::  physical
    COMPLEX, DIMENSION (:, :, :), ALLOCATABLE ::  fourier
    TYPE (Grid) ::  mesh
    LOGICAL ::  temporary
END TYPE Field
```

where we capitalize Fortran keywords as visual cue to code structure and where the components of a Field are its physical-space samples, the corresponding Fourier-space coefficients, the mesh point locations of the physical-space samples, and a Boolean indicator of whether the given Field object is temporary or persistent. We detail the Fourier spectral solution of similar equations in a recently submitted paper [17]. Here we focus on memory management issues only.

The Fortran 95/2003 standards require overloaded operators and the procedures they call to be free of side effects – that is they cannot modify their arguments. The "`INTENT(IN)`" attribute enforces this condition, but also poses a memory management dilemma. In composite (nested) function calls, the `left` and `right` arguments could have

Table 1
Memory leak test results (N.A. = vendor supplied intermediate C code)

Compiler	Platform(s)	Version	Memory Leak?
Absoft	Cray XD1	9.0-1	yes
g95	Windows PC	4.0.2	no
Intel	Linux PC, SGI Altix	9.0	yes
Portland Group	Linux PC, Cray XD1	6.1	yes
NAG	N.A.	N.A.	no

appeared as a `RESULT` in an operator of higher precedence. The operator passing its result to `field plus Field` is likely to have allocated memory for that result's `fourier` or `physical` component just as `field plus Field` will likely need to allocate space for `total%fourier or total%physical` via a statement of the form

```
ALLOCATE(total%physical(nx,ny,nz))
```

where `nx`, `ny`, and `nz` are integers. As several authors have pointed out, the easiest and most efficient place for the programmer to release memory that was dynamically allocated inside the result of one operator is inside the operator to which this result is passed [11,15,19]. However, the operator receiving the result cannot modify it.

A similar dilemma relates to defined assignments such as

```
PUBLIC ::  ASSIGNMENT (=)
INTERFACE ASSIGNMENT (=)
  MODULE PROCEDURE field equals field
END INTERFACE ASSIGNMENT (=)
SUBROUTINE field equals field (left, right)
  TYPE (Field), INTENT (OUT) ::  left
  TYPE (Field), INTENT (IN) ::  right
```

where there frequently arises a need to free memory associated with allocatable components of `right` if it is the result an expression evaluation.

The Fortran standard resolves the above dilemmas by stipulating, "When a variable of derived type is deallocated, any ultimate component that is a currently allocated allocatable array is deallocated (as if by a DEALLOCATE statement)" [10]. Since the compiler is responsible for deallocating `left` and `right` after they go out of scope, the standard obligates compilers to deallocate these variables' allocatable components also. Nonetheless, Table 1 shows three of the five compilers tested exhibited memory leaks for at least some overloaded expressions. We can provide the relevant test code for each compiler upon request.

Even compilers that do the proper deallocations might not do so economically. For example, we have analyzed intermediate code received from the Numerical Algorithms Group (NAG) and found that the NAG compiler carries along all the memory allocated at intermediate steps in the call tree, performing deallocations only after the final assignment at the top of the tree. This extremely lax, although arguably valid, interpretation of the standard could lead programmers working with large arrays to abandon operator overloading until the situation improves.

We seek to facilitate the use of allocatable components now, rather than awaiting the long overdue vendor corrections. We also seek to economize the dynamic memory usage in ways the standard does not require. Achieving these goals requires addressing widespread and longstanding inadequacies in compilers, *not* inadequacies in the standard.

3. Formal specification

3.1. Hermeticity

We can now specify the constraints that preclude a category of memory leaks in composite operator calls. We refer to leak-free execution as hermetic memory management, or simply *hermeticity*. The memory of interest is associated with the allocatable array components inside data structures passed into, through and out of call trees of the form of Fig. 2. In our applications, we find no need for any other large memory allocations inside operators, so we will assume no others take place. This assumption is discussed further vis-à-vis economy in Section 3.2.

Figure 4 details several operators of interest. These include unary operators such as `d dx()`, binary operators such as `OPERATOR(+)` and a defined assignment `ASSIGNMENT(=)`. Our primary task is to constrain the behavior of such operators using the four memory management rules listed in Section 2.1. We model the *Definition* and *Corollary* simply by including a Boolean temporary attribute in the `Field` object interface (see Fig. 1). The value of this attribute classifies the object in the set of objects defined by the *Definition* or in the complementary set defined by the *Corollary*.

In formal methods, constraints take three forms: preconditions that must be true before a procedure starts, postconditions that must be true after it ends, and invariants that must be true throughout its execution. We formally specify *Rule 1* from Section 2.1 through postconditions on the arithmetic and differential operators. The contexts for OCL pre- and postconditions are always operations or methods. In OCL, one writes the constraint context above the constraint. Thus, OCL postconditions are written in the following form:

```
context: Field::
  field plus Field (left: Field, right: Field) total: Field
post: total.temporary = true
```

where the text following "`context:`" provides the class name followed by the operator signature, including the operator name, its argument names and types, and its result name and type. The postcondition following the above "`post:`" label stipulates that all results returned by `field plus field()` must be classified as temporary. The corresponding constraint for unary operators takes the same form.

Rule 2 governs the left-hand arguments to defined assignments:

```
context: Field:: assignment (=) (left: Filed, right: Field)
post: left.temporary = false
```

Although there is no explicit result passed by a defined assignment (Fortran 95/2003 requires a defined assignment to be a SUBROUTINE), the implicit result is the left-hand argument, which is passed by reference and modified to contain a copy of the right-hand argument. The latter postcondition specifies that `left` must be persistent.

Rule 3 governs the deletion of temporary objects. It applies to all operators, including unary and binary ones, along with defined assignments. As applied to the unary operator `d dx`, the corresponding formal constraint takes the form

```
context: Field::
  d dx (s: Field, direction: Integer) ds dx: Field
post: s.temporary implies not s. allocated()
```

Although `d dx()` takes two arguments, we refer to it as a unary operator because the second argument simply determines the coordinate direction in which the first argument will be differentiated. Predicate calculus stipulates that the latter postcondition evaluates to true if the expression after the `implies` operator is true whenever the expression before `implies` is true. The constraint also evaluates to true whenever the expression before the `implies` is false.

Rule 4 governs the deletion of persistent objects. Only defined assignments create persistent objects. Since all `Field` operators have access to private `Field` components, they can make direct assignments to those components. `Field` operators therefore do not call the defined assignment procedure, so most persistent `Field` objects are created outside the `Field` class. An example occurs inside the third-order, Runge-Kutta time-advancement method, `RK3 Integrate`, of the `Integrand` class in Fig. 1. As described by RMX, `Integrand` is a polymorphic class that uses dynamic dispatching to integrate instances of various classes from some time t_n to some time $t_{n+1} = t_n + \Delta t$. The `Integrand` argument passed into `RK3 Integrate` effectively represents a virtual object whose actual type is determined at run time.

During execution, `RK3 Integrate()` instantiates several temporary instances of whatever class of object it receives. Two such instances contain the state variables at the end of the two intermediate Runge-Kutta substeps. Given an `Integrand` T containing a `Scalar` temperature field, the two intermediate instances are typically referred to as T' and T'', so we name them `Tp` and `Tpp`. To write constraints on these objects, we model them as attributes of the `Integrand` class – although they are actually implemented as local variables in `RK3 Integrate()` since they are not currently used by other time advancement methods within the Integrand class.

To formalize *Rule 4*, we specify invariant constraints on the persistence of `Tp` and `Tpp`. For OCL invariants, the context is always a class, an interface, or a type. For the Integrand class, we wright

```
Context:  Integrand
Tp.temporary = false
Tpp.temporary = false
```

Since `RK3 Integrate` instantiates these objects, the formal constraints corresponding to *Rule 4* requires they be deleted upon termination of this procedure:

```
Context:  Integrand ::  RK3 Integrate (k:  Integrand)
post:  Tp.allocated () = false
post:  Tpp.allocated () = false
```

Similar postconditions apply to any other integration procedures that use `Tp` and `Tpp`.

As valid OCL expressions, the above constraints are backed by a formal grammar defined in Extended Backus-Naur Form (EBNF) [16]. An EBNF grammar specifies the semantic structure of allowable statements in a formal language. The statements' meanings can therefore be communicated unambiguously as part of the UML design document. An additional advantage of formal methods is their ability to express statements that could not be written in an executable language. One example is the relationship between the Boolean expressions in the `implies` statement. Another is the fact that these constraints must be satisfied on the set of *all* instances of the class. Programs that run on finite state machines cannot express conditions on sets that must in theory be unbounded. An additional benefit is that the set theory and predicate logic behind formal methods facilitate proving additional desirable properties from the specified constraints. (Such proofs are most naturally developed by translating the constraints into a mathematical notation that is opaque to those who are not formal methods specialists. Since the non-specialists OCL targets coincide with our intended audience, proving additional properties mathematically would detract from our main purpose.) Finally, a more concrete benefit will be explained. in Section 4.2 after detailing the run-time checks inspired by the OCL pre- and postconditions.

The run-time checks inspired by the above OCL pre- and postconditions will be detailed in Section 4.2, where a more concrete benefit will be explained.

3.2. Economy

The above rules can be refined to encourage more economical memory usage. Although the memory reductions are modest, we hope to demonstrate an important tangential benefit from the way OCL forced us to think abstractly about our software model. For this purpose, consider again the function `field plus field()`, which takes arguments `left` and `right`, return `total` and has the generic interface `operator(+)`]. We can ensure economical memory usage by specifying that temporary memory be recycled. To facilitate this, `left`, `right`, and `total` must be pointers. In Fortran 2003, this means adding the `POINTER` attribute to their declarations as follows:

```
FUNCTION field plus Field (left, right) RESULT (total)
  TYPE (Field), POINTER, INTENT (IN) ::  left, right
  TYPE (Field), POINTER ::  total
```

Since OCL does not have a pointer type, we model pointers as a UML association between two classes. We model `left`, `right` and `total` as instances of a `Field Pointer` class that exists solely for its association with the `Field` class. It is assumed here the `Field Pointer` class implements the services of Fortran 95/2003 pointers, including an `ASSOCIATED()` method that returns a Boolean value specifying whether its first argument is associated with its second argument.

In Fig. 5, we apply the label "`target`" to the `Field` at the other end of the association. From a `Field Pointer` object, this association can be navigated through the OCL component selection operator ".", so an economical postcondition might be

```
context:  Field::
  operator (+) (left:  Field Pointer, right:  Field Pointer)
  total:  Field Pointer
post:  left.  target.temporary implies
    associated (total, left.target)
post:  ((not left.  target.temporary) and
    right.  target.temporary) implies
    associated (total, right.target)
```

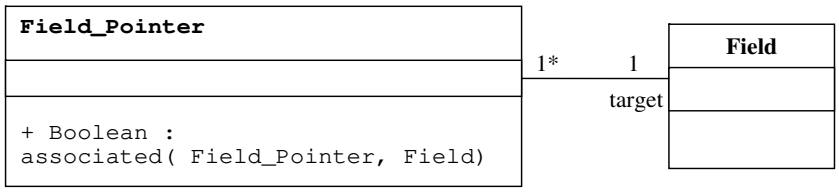


Fig. 5. Field pointer class model (numerical values indicate that one or more Field pointer instances can be associated with any one Field instance).

where the implied conditions stipulate that `total` must be associated with one of the temporary targets. We assume the programmer has overwritten the corresponding temporary with the sum of `left` and `right`, but we leave out this part of the specification since the meaning of the sum is application-dependent. In our application, it simply means that all component real and integer variables and arrays have been summed.

We advocate reusing the entire temporary object, while giving its array component the `ALLOCATABLE` attribute, rather than the `POINTER` attribute. In Fortran 95/2003, an `ALLOCATABLE` array can be thought of as a limited type of pointer – one which can be associated with memory only through an `ALLOCATE` statement, not through target redirection or pointer arithmetic. This limitation has important performance benefits with optimizing compilers. As discussed by Metcalfe, Reid and Cohen [12], for example, `ALLOCATABLE` arrays are guaranteed to contain contiguous memory that can be accessed with unit stride; whereas a Fortran `POINTER` can target an array subsection that might not be contiguous. Any resulting lack of spatial locality could retard cache performance.

For us, the greatest benefit of developing the economizing constraints relates to the conceptual leap required to model pointers in OCL. Prior to making that leap, we had given up on expressing `INTENT` for `POINTER` arguments. The ability to do so is a Fortran 2003 feature not yet available in some compilers, e.g., g95, and only very recently added to others, e.g., Intel. Developing the `Field Pointer` class model inspired us to implement an analogous data structure that facilitated emulating pointer intent before it was available in compilers. This breakthrough enabled operator overloading. Without it, we could not even satisfy the hermeticity constraints of Section 3.1 unless we used an awkward syntax as discussed next.

Recall that Fortran requires operator arguments to have the `INTENT(IN)` attribute, which precludes the deallocations necessary to ensure hermeticity with several current compilers. To circumvent this restriction, we previously evaluated derived type expressions by invoking procedures by name, while writing the operator syntax in an adjacent comment

```
TYPE (Field) ::  left, right, total
!            total =      left + right
CALL assign (total, plus (left, right))
```

where `assign` and `plus` are aliases for `field equals field` and `field plus field`, respectively. To see the tremendous value of operator overloading, consider the RHS of Eq. (1), for which our previous syntax was

```
TTPE (Field) ::  dT dt, T
CALL assign (dT dt, &
minus (times (1/alpha, &
plus (d2 dx2 (T, 1), plus (d2 dx2 (T, 2), d2 dx2 (T, 3)))), &
plus (times (u, d dx (T, 1)), plus (times (T, d dx(T, 2)), &
times (w, d dx (T, 3)))))
```

where ampersands indicate line continuation. The corresponding overloaded syntax is much closer to the form of differential equation being approximated:

```
dT dt = (1/alpha)* (d2 dx2 (T, 1) + d2 dx2 (T,2) + d2 dx2 (T, 3)) & − (u*d dx
(T,1) + v*d dx (T,2) + w*d dx (T, 3))
```

The overloaded syntax requires applying the `INTENT(IN)` attribute to a `Field Pointer` object to follows:

```
TYPE Field Pointer
  TYPE (Field), POINTER ::  target field
END TYPE Field Pointer
```

```
FUNCTION field plus Field (left, right) RESULT (total)
  TYPE (Field Pointer), INTENT (IN) ::  left, right
  TYPE (Field Pointer), ::  total
```

This restricts us from changing with what target object the `target field` pointer is associated, while it allows us to `DEALLOCATE` the `ALLOCATABLE` array inside the target.

4. Discussion and results

4.1. Limitations

Subsequent to the publication of the RMX memory management rules, we found an informal report by Markus [11] and Stewart [19] describing a strategy that, at its core, is algorithmically equivalent to ours. One significant difference between their approach and ours is their use of pointers to allocate the component arrays *inside* abstract data types. We have already discussed the performance advantage of allocatable arrays. We now summarize a limitation of our approach pointed out by Markus and Stewart.

If the ultimate step in the call tree is a Fortran intrinsic command or system call, the programmer cannot release the memory allocated for derived type components at the penultimate step. Markus gives an example equivalent to

```
PRINT *, a + b
```

where `a` and `b` are instances of an abstract data type and "+" is an overloaded operator. Stewart solves this problem by keeping a running tally of the nesting level at which an object is being used. He stores the tally in the object itself and deletes the object at the outermost nesting level. Specifying this in a UML/OCL model requires writing constraints for all procedures in the model, not just overloaded operators. Although we have not needed this technique in our applications, it is clearly of value in some projects.

4.2. Run-time assertion checking

Modern tools exist for automatically generating Java code from OCL. We know of no attempts to extend this capability to Fortran, but it is noteworthy that some of the earliest work on automated insertion of run-time assertions was in Fortran 77 [20].

Run-time checking is necessarily incomplete because each run tests only one set of inputs, but this is of no consequence in our case because the memory allocations required to advance the simulation are the same at each time step independent of the initial data. Appendix A presents Fortran subroutines that approximate our OCL hermeticity constraints. We call `pre Field()` at the beginning of each Field class method (except in constructors). We call `post Field()` at the end of each method (except in destructors). Although the specification presented in Section 3 was slightly simplified in the interest of brevity, the full implementation in Appendix A is largely self-explanatory. Each `Field` instance contains two `ALLOCATABLE` array components: one stores a physical-space representation and another stores the corresponding Fourier coefficients. In our implementation, it is never valid for both to be allocated except momentarily in the routine that transforms one to the other. This condition is checked in the Appendix A code.

Besides monitoring our source code behavior, we found these routines eminently useful in checking for compiler bugs. Using these routines, we discovered that the Portland Group compiler was not always allocating arrays as requested. Despite repeated attempts to reproduce this problem in a simple code to send the compiler vendor, our simplest demonstration of the error was a 4500-line package in which the error only occurs after several time steps inside a deeply nested call tree. The related saga cost us several weeks of effort and has so far taking the vendor over one year repair. The disciplined approach to checking assertions at the beginning and end of each procedure, as inspired by the OCL pre- and postconditions, paid its greatest dividend in motivating us to switch compilers.

Appendix B presents a procedure that verifies economical memory recycling based on the `Field Pointer` class of Fig. 5. Using this code, we have determined that memory allocated for temporary arguments is recycled by associating it with the function result. We also monitored our code's memory utilization with the Linux shell command "top", which displays in real time the processes using the most memory. The memory occupied by our code closely matched our calculation based on recycling and the maximum memory usage remained nearly constant over time, implying hermeticity.

Table 2
Procedural run-time distribution

Function Name	Number of calls	% of total execution time
transform Field	108	34.32
field equals Field	167	19.67
d dx Field	156	16.18
dealiasing uv Field	36	7.70
field plus Field	75	7.43
field times Field	36	5.03
...	...	...
pre Field	1447	0
post Field	885	0

4.3. Performance

Since our pre- and postcondition subroutines contain simple Boolean expressions, they require insignificant amounts of execution time. Table 2 shows our most costly procedure is the `transform Field()`, which contains 3D FFT calls. This procedure accounts for 34% of the processor time. By contrast the constraint checkers `pre Field()` and `post Field()` occupy immeasurably low percentages of execution time even though the number of calls to these routines exceeds the calls to `transform Field()` by roughly an order of magnitude.

We address broader performance issues in a separate paper [17], including opportunities for coarse-grained parallelism in the upper three layers of the class hierarchy in Fig. 1 along with fine-grained parallelism in the lower two layers. In classes where the compilers are leak-free, the operators do not need to modify their arguments, so the procedures can have the Fortran 95 `PURE` attribute, a feature that facilitates automatic parallelization across calls to the operators. Inside the operators, long loops are often replaced with the Fortran 95 `FORALL` construct, a feature that facilitates further automatic parallelization.

5. Conclusions

We have demonstrated that the process of converting previously informal rules into formal constraints written in OCL benefited us in two significant ways. First, the notion of writing pre- and postconditions that must be true before and after a procedure's execution, respectively, enabled us to expose an exceptionally hard-to-find compiler bug that presented itself only after several iterations through 4500 lines of code. Second, the thought process involved in the abstract modeling of an otherwise language-specific construct, pointers, inspired us to encapsulate Fortran 95 pointers in a data structure that we used to emulate a Fortran 2003 feature before it had been added to compilers. In particular, our emulation of the ability to specify pointer intent liberated us from an extremely cumbersome notation by enabling programmer-controlled, hermetic operator overloading.

Although programmer-controlled deallocation of allocatable array components will no longer be necessary once compilers properly implement the Fortran 2003 standard, our techniques enable programmers to use derived types with allocatable components in overloaded expressions now, rather than await vendor compliance. The techniques are not without limitations, but we have found them sufficient for solving several partial differential equations, including the equation for thermal convection and conduction.

Finally, we monitored the run-time memory utilization of our code and found it to be consistent with hermetic, economical execution. We also demonstrated that the run-time assertions inspired by our pre- and post-condition subroutines have negligible cost in terms of execution time.

Acknowledgements

This work was supported in part by Award No. 61-8257-0-6-5 from the Office of Naval Research.

Appendix A. Hermetic Field operator pre- and post-conditions

```
SUBROUTINE pre_Field(this,constructor)
    IMPLICIT NONE
    TYPE(Field) ,INTENT(IN)           :: this
    LOGICAL     ,INTENT(IN), OPTIONAL :: constructor
    LOGICAL :: both_allocated &
              ,at_least_one_allocated &
              ,is_constructor

    is_constructor=.FALSE.

    IF (PRESENT(constructor)) is_constructor=constructor

    at_least_one_allocated  = &
    ALLOCATED(this%fourier) .OR.  ALLOCATED(this%physical)
    both_allocated          = &
    ALLOCATED(this%fourier) .AND. ALLOCATED(this%physical)

    IF (both_allocated) &
      STOP 'pre: constructor result previously allocated'

    IF (is_constructor) THEN
      IF (at_least_one_allocated) &
        STOP 'pre: constructor result partially allocated'
    ELSE
      IF ( .NOT.( at_least_one_allocated ) ) &
        STOP 'pre: invalid argument'
    END IF

  END SUBROUTINE pre_Field

  SUBROUTINE post_Field(this,public_operator,deletable)
    IMPLICIT NONE
    TYPE(Field) ,INTENT(IN) :: this
    LOGICAL     ,INTENT(IN) :: public_operator
    ! => could appear in a derived type expression
    LOGICAL     ,INTENT(IN) :: deletable
    ! => must be deallocated if temporary
    LOGICAL                 :: at_least_one_allocated &
                               ,both_allocated

    at_least_one_allocated  = &
    ALLOCATED(this%fourier) .OR.  ALLOCATED(this%physical)
    both_allocated          = &
    ALLOCATED(this%fourier) .AND. ALLOCATED(this%physical)
    IF (both_allocated) STOP  post: invalid result

    IF (public_operator .AND. deletable) THEN
       IF (this%temporary) THEN
         IF (at_least_one_allocated) &
           STOP 'post: invalid operation on temporary'
       ELSE
         IF (.NOT.at_least_one_allocated) &
           STOP 'post: invalid operation on non-temporary'
       END IF
    ELSE
       IF (.NOT.at_least_one_allocated) &
         STOP 'post: invalid operation result'
    END IF

  END SUBROUTINE post_Field
```

Appendix B. Economical Field operator post-condition

```
SUBROUTINE economical_Field(left,right,result)
  IMPLICIT NONE
  TYPE(Field_Pointer) ,INTENT(IN) ::left,right,result

  IF (left%target%temporary) THEN
     IF (.NOT. ASSOCIATED(result%target,left%target) &
       STOP 'post: left argument not recycled'
  ELSE IF (right%target%temporary THEN
     IF (.NOT. ASSOCIATED(result%target,right%target) &
       STOP 'post: right argument not recycled'
     END IF
  END IF

END SUBROUTINE economical_Field
```

References

[1] E. Akin, *Object-oriented Programming via Fortran 90/95*, Cambridge University Press, Great Britain, 2003.

[2] J.J. Barton and L.R. Nackman, *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*, Addison-Wesley 1994.

[3] P. Bientinesi, J.A. Gunnels, M.E. Myers, E.S. Quintana-Orti and R.A. van de Geijn, The science of deriving dense linear algebra algorithms, *ACM Transactions on Mathematical Software* **31**(1) (2005), 1–26.

[4] S.M. de Bruyn Kops and J.J. Riley, Direct numerical simulation of laboratory experiments in isotropic turbulence, *Phys Fluids* **10**(9) (1998), 25–27.

[5] L.A. Clarke and D.S. Rosenblum, A historical perspective on runtime checking in software development, *ACM SIGSOFT Software Engineering Notes* **31**(3) (2006), 25–37.

[6] V.K. Decyk, C.D. Norton and B.K. Szymanski, How to express C++ concepts in Fortran 90, *Scientific Programming* **6** (1997), 363–390.

[7] V.K. Decyk, C.D. Norton and B.K. Szymanski, How to support inheritance and run-time polymorphism in Fortran 90, *Computer Physics Communications* **115** (1998), 9–17.

[8] R. van Engelen, L. Wolters and G. Cats Tomorrow's weather forecast: Automatic code generation for atmospheric modeling, *IEEE Computational Science and Engineering*, July-September 1997.

[9] L. Hatton, The T Experiments: Errors in Scientific Software, *IEEE Computational Science & Engineering* **4**(2) (1997), 27–38.

[10] J3 Fortran Standards Technical Committee, *Technical Report ISO/IEC 15581:1998(E)* International Organization for Standards/International Electrotechnical Committee, Geneva, Switzerland, 1998.

[11] A. Markus, Avoiding memory leaks with derived types, *ACM Fortran Forum* **22**(2) (August 2003).

[12] M. Metcalfe, J. Reid and M, Cohen, *Fortran 95/2003 Explained*, Oxford University Press, Oxford, 2004.

[13] J.N. Oliveira, 1997, http://www.di.uminho.pt/~jno/html/cam-wfm.html.

[14] D.K. Pace, Modeling and simulation verification challenges, *Johns Hopkins APL Technical Digest* **25**(2) (2004), 163–172.

[15] D.W.I. Rouson, K. Morris and X. Xu, Dynamic memory de-allocation in Fortran 95/2003 derived type calculus, *Scientific Programming* **13**(3) (2005).

[16] D.W.I. Rouson and Y. Xiong, Design metrics in quantum turbulence simulations: How physics influences software architecture, *Scientific Programming* **12** (2004), 185–196.

[17] D.W.I. Rouson and X. Xu, A grid-free abstraction of the Navier-Stokes equations via data structure wrappers in Fortran 95/2003, submitted to *ACM Transactions on Mathematical Software* in March 2006.

[18] D.E. Stevenson, How goes CSE? Thoughts on the IEEE CS Workshop at Purdue, *IEEE Computational Science & Engineering* (April–June 1997).

[19] G.W. Stewart, Memory leaks in derived types revisited, *ACM Fortran Forum* **22**(3) (December 2003).

[20] L.G. Stucki and G.L. Foshee, New assertion concepts for self-metric software validation, *Proceedings of the International Conference on Reliable Software*, ACM SIGPLAN and SIGMETRICS, Los Angeles, California USA, 1971, 59–71.

[21] J. Warmer and A. Kleppe *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed., Addison-Wesley, New York, NY, USA 2003.