Scientific Programming 16 (2008) 329–339
DOI 10.3233/SPR-2008-0262
IOS Press

Towards analysis-driven scientific software architecture: The case for abstract data type calculus

Damian W.I. Rouson
Scalable Computing Research and Development Department, Sandia National Laboratories, MS 9158, P.O. Box 969, Livermore, CA 94550, USA

Abstract. This article approaches scientific software architecture from three analytical paths. Each path examines discrete time advancement of multiphysics phenomena governed by coupled differential equations. The new object-oriented Fortran 2003 constructs provide a formal syntax for an abstract data type (ADT) calculus. The first analysis uses traditional object-oriented software design metrics to demonstrate the high cohesion and low coupling associated with the calculus. A second analysis from the viewpoint of computational complexity theory demonstrates that a more representative bug search strategy than that considered by Rouson et al. (*ACM Trans. Math. Soft.* **34**(1) (2008)) reduces the number of lines searched in a code with λ total lines from $O(\lambda^2)$ to $O(\lambda \log_2 \lambda)$, which in turn becomes nearly independent of the overall code size in the context of ADT calculus. The third analysis derives from information theory an argument that ADT calculus simplifies developer communications in part by minimizing the growth in interface information content as developers add new physics to a multiphysics package.

Keywords: Objected-oriented programming, design metrics, complexity, information theory, Fortran

1. Introduction

The central activity in object-oriented (OO) software design involves decomposing a problem into a set of abstract data types (ADTs). These abstractions encapsulate private data tightly coupled to public procedures that operate on that data. A recurring strategy across numerous scientific software projects over the past decade was to choose ADTs that closely mimic continuous mathematical abstractions such as scalar, vector, and tensor fields [5,7,10,15]. By further defining a set of algebraic, integral, and differential operators, such strategies support an ADT calculus that hides discrete numerical algorithms behind interfaces to continuous ones. As noted on the Sundance project home page, this strategy lets software abstractions resemble blackboard abstractions.[1] One expects the result to be programs that exhibit a degree of elegance in some sense, which naturally generates two fundamental questions: to what degree and in what sense? This article presents steps towards answering these questions.

[1] http://www.math.ttu.edu/~klong/Sundance/html/.

The stated aim requires developing quantitative, analytical ways to describe scientific software architectures. Attempts to do so in the broader software engineering community have met with some controversy [3,17,20]. Much of the controversy surrounds the utility of software design metrics as:

1. Fault rate predictors,
2. Programmer productivity evaluators, and
3. Development time estimators.

For example, in programs with modular architectures, one might expect fault rates to increase monotonically with module size; yet Fenton and Ohlsson [2] detected nearly constant fault rates independent of module size in their study of two releases of a major telecommunications software project. In fact, the only significant departure from this behavior occurred at the smallest module size, where the fault rates dropped precipitously in one release and spiked upwards in the subsequent release. They hypothesized that the latter behavior stemmed from the increased ratio of interface content to body content for the smallest modules, so that complexity in the body had been pushed into the interface.

As tools for evaluating productivity, design metrics suffer from potential manipulation by programmers and misinterpretation by their managers. To use the simplest example, were one to use the much maligned source lines of code (SLOC) as the measure of productivity, even novice programmers could manipulate the result by inserting dead code or splitting steps that could easily be combined. On the manager's side, a tendency to task the more experienced developers with the more difficult problems might yield the lowest SLOC for the most adept people.

The use of design metrics to estimate development time provokes possibly the most pessimistic backlash. For example, Lewis [9] argued that neither a program's size nor its complexity can be objectively estimated *a priori* – and likewise for its development time and the absolute productivity of its developer. Lewis backed these claims with algorithmic complexity theory. The basic argument regarding program size and complexity follows: for a string s, the associated algorithmic complexity $K(s)$, defined as the shortest program that produces s, is provably non-computable. Taking s to be the desired program output suggests that the smallest program (including input data) that produces that output cannot be estimated. Alternatively, easily derived upper bounds on program size, such as the size of a program that simply prints the output string, do not prove useful. The lack of useful, computable bounds on program size implies a similar statement for development time since it depends on program size as well as a similar statement for programmer productivity as measured by program size divided by development time. Ultimately, Lewis concludes that even approximate estimations cannot be constructed theoretically and must therefore be judged purely on empirical grounds.

None of the above precludes calculating algorithmic complexity, or other formal properties of software, within restricted domains wherein certain program properties can be guaranteed. For example, Chaitin [1] demonstrated "how to run algorithmic information theory on a computer" by restricting the program to being expressible in so-called "pure LISP". In this context, one assumes programs to be self-delimiting – that is to contain information about their lengths. In such a system, one can construct an arbitrarily large number of digits of the program's algorithmic complexity.

Somewhat ironically, Kirk and Jenkins [8] provided *empirical* evidence of the usefulness of the very entity, algorithmic complexity, that Lewis employed in arguments against *theoretical* estimates. They took the concatenation of a compressed piece of source code and the compression software that produced it as an upper bound on $K(s)$. Furthermore, they used code obfuscators to estimate the worst-case complexity and compare this with the complexity added by the developer during a revision cycle. Obfuscators alter source code in ways that make it more difficult for humans to understand and more difficult to reverse engineer. Using the compression software bzip2,[2] they estimated the character symbol entropy of obfuscated files at varying levels of obfuscation. Based on their results, they proposed analogies between their information entropy measures and thermodynamic entropy. They further suggested that these analogies could lead to studying material-like phase transitions as software becomes more brittle through stronger coupling between modules.

A common activity in computational applications of complexity theory is the derivation of so-called *polynomial time* estimates for the completion of various tasks. Inspired in part by this work, Rouson et al. [14] derived a polynomial time estimate for the completion of one process that impacts productivity: the bug search. This approach draws additional inspiration from Shalloway and Trott [18] and Oliveira and Stewart [12], who made extremely similar comments in two very different application domains: OO design patterns and procedural scientific programming. Both pairs of authors commented that programmers typically spend more time searching for bugs than fixing them. Combining an artificially constructed bug search algorithm with empirical data on statically detectible fault rates in scientific C and Fortran 77 programs [6], Rouson et al. [14] demonstrated that the time to find all bugs in procedural programs with globally shared data grows quadratically with SLOC. In this context, global data sharing might result from using Fortran COMMON blocks or simply from passing solution vectors between procedures instead of hiding them behind ADT interfaces.

Rouson et al. [14] further demonstrated that constructing an ADT calculus renders the aforementioned quadratic polynomial roughly constant by freezing the relevant SLOC at its value for a single ADT and then holding SLOC nearly constant across ADTs as the software project grows. The current paper builds upon that work. Section 2 provides context by discussing a target set of applications. In these applications, one desires to construct an ADT calculus for purposes of

[2] http://www.bzip.org.

writing coupled ordinary and partial differential equation solvers. Section 3 analyzes such a calculus. Section 3.1 discusses two object-oriented design metrics: coupling and cohesion. Section 3.2 broadens the applicability of the bug search metric by analyzing a more realistic search algorithm: the bisection method. Section 3.3 develops novel information theoretic arguments that offer the first opportunity to quantify another development-time process: developer communications. Section 4 summarizes the conclusions from this study.

2. Problem formulation and methodology

2.1. Discrete time advancement

The problems of interest for the current work involve solving coupled sets of linear and nonlinear, ordinary differential equations:

$$\frac{\mathrm{d}}{\mathrm{d}t}\vec{V} = \vec{f}(\vec{V}), \tag{1}$$

where $\vec{V} = (V_1, V_2, \ldots, V_N)^{\mathrm{T}}$ is a N-dimensional solution vector. In practice, this vector might comprise the complete state space of some dynamical system or it might represent a finite-dimensional approximation to an infinite-dimensional system such as a set of coupled partial differential equations (PDEs). For example, the elements of $\vec{V}$ might be nodal values at the vertices of a multidimensional grid overlaying the solution domain for a set of PDEs.

Discrete time advancement schemes make various approximations to the integration of Eq. (1) over a small, finite time step $\Delta t \equiv t_{n+1} - t_n$:

$$\vec{V}^{n+1} = \vec{V}^n + \int_{t_n}^{t_{n+1}} f(\vec{V})\,\mathrm{d}t, \tag{2}$$

where $\vec{V}^n$ and $\vec{V}^{n+1}$ are the solution vector at times t_n and t_{n+1}, respectively. One can derive most commonly employed numerical integration algorithms from Eq. (2) by an appropriate choice of a polynomial approximation for the components of $\vec{f}$. For example, first- and second-order Runge–Kutta schemes result from constant and linear approximations interpolating each component of $\vec{f}(\vec{V}^n)$ and $\vec{f}(\vec{V}^{n+1})$. The first-order scheme is:

$$\vec{V}^{n+1} = \vec{V}^n + f(\vec{V}^n)\Delta t, \tag{3}$$

while the second-order scheme is:

$$\vec{V}^{n+1/2} = \vec{V}^n + f(\vec{V}^n, t)\frac{\Delta t}{2} \tag{4a}$$

$$\vec{V}^{n+1} = \vec{V}^n + f\left(\vec{V}^{n+1/2}, t + \frac{\Delta t}{2}\right)\Delta t. \tag{4b}$$

Additionally, one can derive multistep explicit (Adams–Bashforth) and implicit (Adams–Moulton) schemes by judicious choices of polynomial order and interpolation points.

2.2. Constructing an ADT calculus

A broad swath of scientific simulation problems fall under the heading of multiphysics modeling, that is the coupled integration of multiple interdependent mathematical models from related physics sub-disciplines. Rouson et al. [15] outlined a set of domain-specific design patterns targeting such applications. Figure 1 depicts a Unified Modeling Language (UML) class diagram using their Puppeteer and semi-discrete model patterns to construct a model for solid particle transport in electrically conducting, magnetohydrodymic (MHD) flows such as the dusty plasmas found in nuclear fusion tokamaks and the semi-solid dispersions found in liquid metal processing. In such settings, the `particle`[3] class likely solves a drag law for discrete particle positions and velocities. The fluid class solves the Navier–Stokes questions for the fluid velocity and pressure fields, and the `magnetofluid` class solves the magnetic induction equation, a form of Maxwell's equations, for the magnetic vector field.

The `dusty_plasma` class plays the role of the puppeteer pattern [15], manipulating a set of puppet classes by delegating all defined arithmetic to those puppets and mediating all communication between them. For example, `dusty_plasma` queries the particles for their spatial locations. It then queries the `magnetofluid` for its velocity at those locations and passes the response back to the particles for calculation of each particle's drag. Finally, `dusty_plasma` queries the particles for their drag forces and passes the response to the `magnetofluid` which can then incorporate an equal and opposite reaction force into its Navier–Stokes calculation.

The ***`integrable_model`***[4] class holds an abstract type containing a set of deferred-binding, type-bound[5]

[3] We use the Courier New font for code excerpts.

[4] In keeping with the UML standard, we use italicized boldface type for abstract classes.

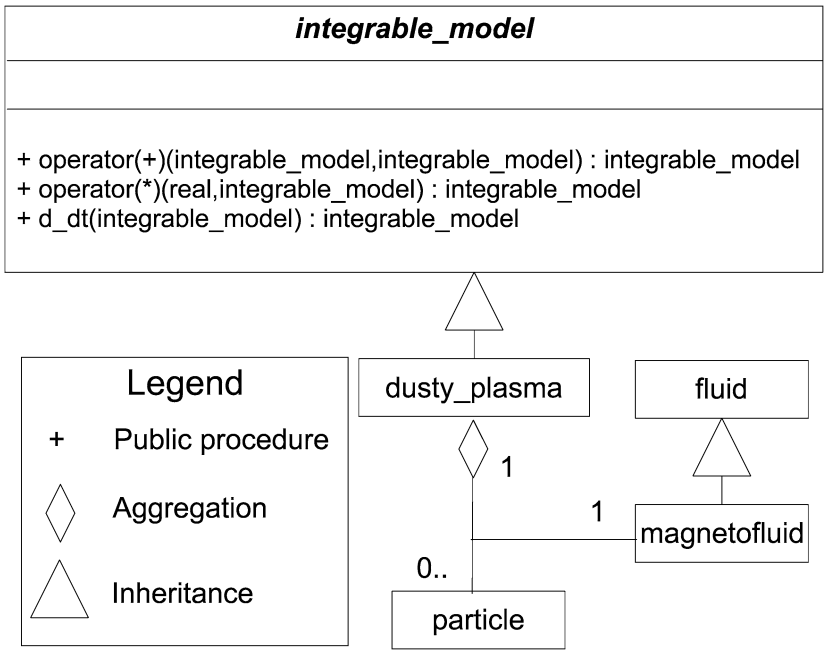

Fig. 1. UML class model for a sample multiphysics application: a dusty plasma that aggregates zero or more particles and one magnetofluid. Numbers near connectors indicate the multiplicity of instances at that end of the connection with "0.." indicating zero or more. Italicized boldface indicates an abstract type.

procedures corresponding to a set of abstract interfaces. These procedures specify a set of algebraic and differential operators for use in the `integrate` procedure, which accepts a dynamically typed argument that it advances one time step using defined operators.[6] The Appendix contains a sample implementation of this class using the new object-oriented constructs of Fortran 2003. For convenience, the `integrate` procedure is also provided here:

```
subroutine integrate(integrand,dt)
  class(integrable_model) :: integrand
  real, intent(in) :: dt
  integrand = integrand &
              + integrand%d_dt()*dt
end subroutine
```

where `integrand` represents a polymorphic entity whose actual type is resolved at runtime and required to be an extension[7] of the `integrable_model` abstract type and where the derived type component selector "`%`"[8] is used to invoke the type-bound procedure `d_dt()`.

The fourth line in the `integrate()` procedure takes advantage of the defined operators "+" and "*", the defined assignment "=", and the time differentiation method "`d_dt()`", all of which must be implemented by any class that extends `integrable_model`. The time differentiator `d_dt()` evaluates $\vec{f}(\vec{V})$ in Eq. (1). The same defined operators, assignments and type-bound procedures could be used to write any order Runge–Kutta method. The second-order method of Eqs (4a) and (4b), for example, could be written as:

```
integrand_half &
  = integrand &
    + integrand%d_dt()*(dt/2.)
integrand &
  = integrand &
    + integrand_half%d_dt()*dt
```

[5] When encapsulated in a Fortran module construct, Fortran 2003 derived types are roughly equivalent to C++ classes and abstract types are analogous to C++ virtual classes where deferred-binding type-bound procedures correspond approximately to virtual member functions.

[6] Fortran defined assignments and operators correspond approximately to what most other OO languages refer to as "overloaded operators".

[7] Fortran 2003 type extension represents Fortran's inheritance mechanism, sharing some features with Java such as the prohibition against multiple inheritance and some features with C++ such as the inheritance of overloaded operators.

[8] The Fortran component selector "`%`" corresponds to the "." member selector in most OO languages.

where `integrand_half` would also be required to be a type that extends the `integrable_model` type.

Any class that implements the desired operators, assignment and differentiation method thereby supports an ADT calculus. In one sense, the definition of such a calculus might be thought of as the construction of a domain-specific language. Given a set of single-physics classes such as `particle`, `fluid`, and `magnetofluid`, application programmers can rapidly assemble an appropriate puppeteer along with new time integrators based on the provided operators. The internal details of each single-physics model would likely involve additional arithmetic and differential operators to calculate the time derivative `d_dt()`. For example, that process might involve calculating various spatial derivatives, integrals, sums, products and differences between the dependent variables of each class, thus expanding the calculus and the types of variables on which it operates [14].

The remainder of this article analyzes ADT-calculus-based discrete time advancement in terms of metrics inspired from three domains: OO design, computational complexity theory and information theory. The aforementioned particle-laden MHD solver provides a context for these analyses and serves as a prototype multiphysics problem. Each analysis addresses the question of scalable development in the sense of scaling up the number of single-physics subsystems in some multiphysics simulation. The analyses focus primarily on the high-level code architecture, treating only briefly any details internal to a specific class or procedure. The reader is directed to references [14–16] for addtional implementation details, including sample code.

3. Analysis

3.1. Object-oriented design metrics

The proponents of OO design metrics aim to improve software engineering practices by encouraging code designs that are coherent, flexible, and robust. In many regards, metrics share these goals with OO design patterns, which can be defined as common solutions to recurring problems in OO software construction. Not surprisingly then the adoption of patterns has a tendency to foster trends in certain metrics. Specifically, many design patterns generate high *cohesion* within an ADT and low *coupling* between ADTs.

The term "cohesion" describes the extent to which an ADT's procedures relate to each other in purpose. Stevens et al. [21] ranked various types of cohesion from weakest to strongest. Coincidental cohesion, the weakest form, occurs when parts of a system have no significant relationship to each other. Functions in classical, procedural mathematics libraries exhibit coincidental cohesion in that most procedure pairs have in common only that they evaluate mathematical functions. Functional cohesion, the strongest form, occurs when each part of a system contributes to a single task.

In the discrete time-advancement problem, each of the operators and type-bound procedures contributes to time advancement. This guarantees functional cohesion at least in a minimal ADT that implements only the requisite calculus.

The term "coupling" describes the extent to which different ADTs depend on each other. As with cohesion, one can rank types of coupling from loosely coupled to tightly coupled. Authors typically rank data coupling as the loosest form [13,17]. Data coupling occurs when one part of a system depends on data from another part. Control coupling, a moderately tighter form, arises when the logical flow of execution in one part depends on flags passed in from another part. Content coupling, which typically ranks as the tightest form of coupling [13,17], occurs when one violates an abstraction by providing one ADT with direct access to the state or control information inside another ADT. An especially egregious form of content coupling can occur when it is possible to branch directly into the middle of a procedure inside one ADT from a line inside another.

Since a developer can write the time advancement expressions in Section 2.2 without any information about the state or flow of control inside the objects and operators employed, the construction of an ADT calculus allows for strict adherence to data privacy. Adopting this practice precludes content coupling. Furthermore, in the discrete time advancement codes of Section 2.2, the only procedure arguments are instances of the class[9] being advanced. These are implicit arguments that arise, for example, when the compiler resolves the statement

```
integrand = integrand &
          + integrand%d_dt()*dt
```

into a procedure call of the form[10]

[9]This article uses "class" and "abstract data type" synonymously.

[10]The ampersand (&) is the Fortran line continuation character character.

```
call assign &
  (integrand,add(integrand,
   multiply(integrand%d_dt(),dt)))
```

where `assign()`, `plus()` and `multiply()` are the defined assignment, addition and multiplication procedures, respectively. Thus, in the absence of any flags internal to the derived type itself, no control coupling arises when employing an ADT calculus. This leaves only the loosest form of coupling: data coupling. At least in terms of the top-level semantics, even data coupling is kept to a minimum given that all operators are either unary or binary, so at most two arguments can be passed.

Inside the time differentiator `d_dt()`, some opportunities arise for minimal control coupling. One is likely to pass a coordinate direction to any partial derivative operators and then to branch inside the derivative procedure based on the passed direction. Even this, however, could be eliminated by defining separate functions for the directional derivatives in each coordinate direction. Thus, it appears the construction of an ADT calculus leads to the desired state of low coupling and high cohesion.

3.2. A complexity-theoretic metric

One branch of computational complexity theory involves deriving operation counts or equivalent time estimates for the completion of various computing tasks. Search algorithms comprise a commonly studied task. Many of the resulting estimates take the form of polynomials in some measure of the problem size such as the number of items in the list being searched. Rouson et al. [14] applied this approach to the bug search process, estimating that the average number of lines searched to find all bugs in a traditional, procedural code λ lines long with globally shared data:

$$\lambda_{searched} = r\lambda\frac{1}{2}\left(\frac{\lambda}{2} - 1\right), \tag{5}$$

where r is the expected defect rate measured in defects per line. Defect rates have been determined empirically in commercially released scientific software [6] and non-scientific software [1].

The $r\lambda$ factor in equation (5) represents the expected number of defects in the entire project, while the remaining factor represents the estimated number of lines searched to find a single defect. The latter quantity resulted from analyzing a chronological code listing wherein every line that executes before the symptom of a problem presents itself is listed before the symptomatic line. With the symptom occurring at $\lambda/2$ on average, causality confines the bug to the preceding $\lambda/2 - 1$ lines. The bug search algorithm involves tracing backwards from this line until the bug is found. On average, this search will terminate at line $(\lambda/2 - 1)/2$.

The aforementioned bug search algorithm has the virtue of being easy to analyze but the vice of not resembling what most programmers do. A more practical search algorithm might resemble the bisection method [4]:

1. Bracket the suspected offending code with statements that verify the satisfaction of necessary constraints. In the most rudimentary cases, the verification might simply involve visual inspection of output. A properly chosen set of constraints will be satisfied at the beginning of the code segment but not at the end.
2. Bisect the code segment and insert a verification statement at the midpoint.
3. Form the next segment from the code between the midpoint and the end point at which the result is the opposite of the midpoint result. For example, if the verification passes at the midpoint, the next segment lies between the midpoint and the end point.

This process can be repeated until the segment is reduced to a single line of code. That line is the first place where the necessary constraint is violated. Figure 2 illustrates two iterations of the bisection bug search method starting from an initial segment encompassing all λ lines of the code.

The number of lines λ_n in the nth segment so-constructed provides a measure of the maximum distance between the lines being tested (the end points nth segment) and the bug. Since the bisection methods cuts this number in half at each iteration, the maximum distance from the bug at iteration n is:

$$\lambda_n = \frac{\lambda}{2^n}, \tag{6}$$

where the initialize segment size has been set equal to the entire source code length λ. Such a bug search converges when the code segment reaches a single line, which happens after

$$n = \log_2 \lambda \tag{7}$$

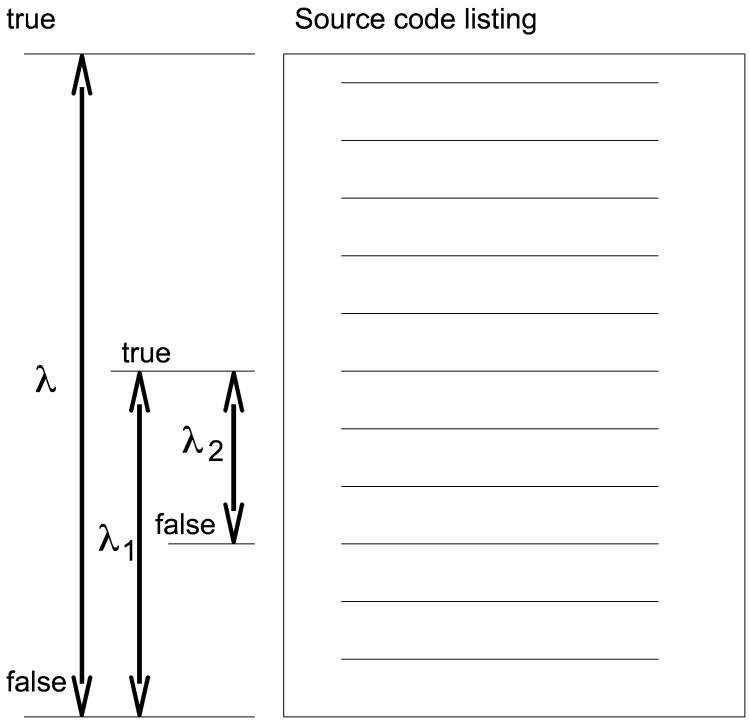

Fig. 2. Bisection bug search algorithm: "true" indicates the location of a successful verification; "false" indicates the location of a failed verification.

iterations. At the beginning of the bisection search ($n = 0$), two lines are checked, those being the first and last lines of the suspect code. At each subsequent iteration, one new line is checked, that being the midpoint line. Thus, the search for all bugs terminates after

$$\lambda_{searched} = r\lambda(2 + \log_2 \lambda) \tag{8}$$

lines, or asymptotically $O(\lambda \log_2 \lambda)$ lines.

Given sufficient insight into the structure of the source code, the programmer can likely make an informed guess about the bug location and thereby reduce the above estimate by starting with a code segment significantly shorter than the entire program length. An important example presents itself in simulation code based on an ADT calculus. With strict adherence to the object-oriented philosophy of data privacy, only the λ_{class} lines in a given class have access to the data that class encapsulates. Any erroneous values observed in private data must have been set by one of the p procedures in that class. This very nearly limits to the search to those procedures. Although occasional excursions outside this class are required to understand the flow of control and ensure that correct data are being supplied to the class, the author's experience suggests these excursions are short and simple due to the simplicity of the expressions that generate the call tree and their close resemblance to the mathematical expressions from which they are derived.

Following Rouson et al. [14], it proves useful to factor λ_{class} into the product of the number of procedures p in the class and the average procedural line density ρ defined so that:

$$\lambda_{class} \equiv \rho p. \tag{9}$$

Two reasons motivate this factoring. First, reviewing Fig. 1 shows that each class that describes a single-physics abstraction extends the ***integrable_model*** abstract class and, therefore, implements the same procedures: a scalar/object multiplication operator "*", an object/object addition operator "+", a defined assignment "=", and a time differentiator "`d_dt`". This holds p constant at 4 across the project even as the number of classes grows to incorporate new physics.

Second, the algorithm implemented inside many of the above operators is very nearly the same. For each class, the multiplication operator multiplies an object's state vector by a constant, while the "+" operator adds

the state vectors of two objects, and the assignment operator "=" copies, or in some cases overwrites, an object's state vector. These statements must be true for each class for the time advancement code from the previous section to work. Since the algorithm is the same for each class, the number of lines required to express the algorithm will be very nearly the same. Hence, ρ remains very nearly constant across the project. This leads to rewriting Eq. (8) as:

$$\lambda_{searched} = r\rho p[2 + \log_2(\rho p)] \tag{10}$$

and to the assertion that $\lambda_{searched}$ very nearly represents a constant independent of the size of the overall project.

3.3. Information-theoretic metrics

Information theory offers a third avenue for analyzing an ADT calculus. In his seminal paper on on the information theory, Shannon [19] quantified the information content and the limits thereof for telecommunication signals. It seems reasonable then to build upon Shannon's foundation to describe the content in other types of communications such as communications between developers. Of interest here is the amount of information developers communicate in supplying new classes that extend the capability of a multiphysics framework of the type described in Fig. 1. In OOP, the minimum information each developer must provide is the interface to that developer's new class. At the top layer in the class model of Fig. 1, `integrable_model` mandates the minimum interface that `dusty_plasma` must implement in order to be integrated forward in time.

Shannon [19] reasoned as follows about the set of all possible messages that can be transmitted between two locations (two developers in the present case):

> If the number of messages in the set is finite then this number or any monotonic function of this number can be regarded as a measure of the information produced when one message is chosen from the set, all choices being equally likely.

He went on to choose the logarithm as the monotonic function because it satisfies several constraints that match our intuitive understanding of information. Consider his expression for the information entropy:

$$H \equiv -\sum_{i=1}^{N} p_i \log p_i, \tag{11}$$

where p_i is the frequency of occurrence of the ith token (e.g., a character or keyword) in some large sample representative of the domain of interest and where N is the number of unique tokens. Taking all choices (tokens) as equally likely leads to $p_i = 1/N$ and, therefore,

$$H = \log N. \tag{12}$$

Now consider again the following two lines from the time integration code of Section 2.2 and Appendix:

```
class(integrable_model) :: integrand
```

and

```
integrand = integrand &
            + integrand%d_dt()*dt
```

The dynamic type `integrable_model` must be resolved to an actual type at runtime. That actual type must extend the parent `integrable_model` type and implement each of the parent's deferred-binding type-bound procedures. An example of such an actual type using the dusty plasma application of Fig. 1 follows:

```
type, extends(integrable_model) :: &
    dusty_plasma
  private
  type(particle) dimension(:), &
    allocatable:: particlePuppet
  type(magnetofluid)          , &
    allocatable:: plasmaPuppet
contains
  procedure, public :: d_dt &
    => dDustyPlasma_dt
  procedure, public :: add &
    => add_dusty_plasma
  procedure, public :: multiply &
    => multiply_dusty_plasma
  procedure, public :: assign &
    => assign_dusty_plasma
end type atmosphere
```

If `dusty_plasma` is the first actual type to extend `integrable_model`, then there exists no ambiguity in the meaning of the two time integration lines above. With only one possible interpretation, $N = 1$, $H = 0$, and the developer communicates no new information. She simply makes the code sufficiently complete to compile and run.

Rouson et al. [14] used an example in which a class named `atmosphere` extended `integrable_`

`model`. Were this to be a second class available in addition to `dusty_plasma`, then atmosphere would increase N to 2. Choosing base 2 for logarithm (which is equivalent to choosing the units in which information content is expressed), the `atmosphere` developer has added $\log_2 2 = 1$ bit of information. More generally, any developer adding a new `integrable_model` descendant adds $\log_2(N+1) - \log_2(N) \equiv \log_2((N+1)/N)$. Furthermore, since the number of classes is countable, N must be an integer, so the jump from $\log_2(N)$ to $\log_2(N+1)$ represents the minimum amount of new information possible. This reflects the ease with which new classes are added to the framework at the highest levels of abstraction.

At lower levels such as the `particle` and `magnetofluid` class, the interfaces remain very similar to that specified at the top level because the puppeteer delegates most type-bound procedure calls to its puppets. The one exception relates to the role the puppeteer places in mediating inter-abstraction communications. That shows up in the interfaces of the puppets as additional arguments that must be passed to the `d_dt()` procedure of a given puppet and as type-bound procedures on other puppets that supply the data for those arguments. Nonetheless, the frequency of occurrence of the various keywords, keyword patterns and even generic procedure names (for example, the operator symbols and names) will be quite high. This ultimately suggests that the growth in information entropy H will typically be very slow as new single-physics classes are added as well.

4. Discussion

While the author believes the aforementioned arguments to be quite strong in favor of ADT calculus, prudence suggests examining some of the pitfalls as well. The outlined approach relies heavily on defined assignments and operators. It has frequently been noted in the literature that operator overloading is "syntactic sugar" – that is a fundamentally unnecessary construct that can always be resolved into a set of non-overloaded direct procedure invocations as described in Section 3.1. The author finds this argument unpersuasive since this can be said of most programming constructs. All high-level programming languages are syntactic sugar that could be, and ultimately are, resolved into machine language instructions. The more compelling question is what are the tradeoffs in using a particular notation. References [14,15] describe some of the performance and memory management tradeoffs of ADT calculus.

One tradeoff with operator overloading is that one must balance the seemingly transparent notation with the ultimately opaque computation it can be used to mask. One can insert computations that bear little resemblance to one's intuitive understanding of the operator's symbol. The author has encountered the need for such computations in trying to write memory-leak-free code using defined assignments and operators in earlier generations of Fortran 95 compilers. Those compiler generations did not have final procedures (known as destructors in some languages) or garbage collection. In the absence of these constructs (and in the presence of some compiler-induced memory leaks), one must resort to schemes that tag certain objects as temporary (ripe for deletion), while tagging other objects as persistent (protected from deletion) [16]. The way such tags are handled inside defined assignments and operators bears no resemblance to the usual meaning of the operators outside of this context. Although necessary in this case to avoid fatal build-up of dynamically allocated memory, this type of programming seems best avoided whenever possible. Fortunately, with the new automatic memory management features and the new final procedure construct in Fortran 2003, at least the above manual memory management is no longer needed.

Finally, although this article emphasizes slightly more rigorous analyses of software architecture than is common in scientific software development, even design metrics that appear to be more sophisticated than SLOC are not necessarily so. Musa et al. [11] showed that most of the complexity metrics that had been studied up to 1986 exhibited a high correlation with SLOC.

5. Conclusions

Three avenues have been explored for analyzing an increasingly common approach to object-oriented scientific programming. That approach involves the use of defined assignments and operators to construct an abstract data type calculus. Such a calculus facilitates the writing of software expressions that closely mirror the blackboard expressions from which they are derived.

The dusty plasma class model provides a setting within which to study an ADT calculus. Since that class model was inspired by design patterns, the first avenue of exploration involved two OO design metrics commonly targeted by patterns: cohesion and coupling. In this setting, the calculus generates the desirable result of high cohesion and coupling.

From the vantage point of computational complexity theory, the bisection method reduces the number of lines searched in a code with λ total lines from $O(\lambda^2)$ in a previous study to $O(\lambda \log_2 \lambda)$. Furthermore, the latter estimate becomes nearly independent of the overall code size in the context of using an ADT calculus to add new physics to a multiphysics simulation package.

Within the framework of information theory, an ADT calculus can be seen to simplify developer communications. It does so by demonstrating that the the information a new developer communicates by extending the abstract base type to create a new puppeteer (multiphysics aggregator and mediator) is one bit. Additionally, the amount of information that must be communicated to add new single-physics models an existing multihphysics package proves quite small since much of the interface is the same for each class except for the coupling terms and the type-bound procedures that compute them.

Finally, this article addresses some of the arguments against of constructing an ADT calculus. While operator overloading is often referred to as "syntactic sugar" that can be otherwise represented without a special language facility, this article argues that this not a compelling criticism insofar as similar statements can be made about every construct in a high-level language. Likewise, while acknowledging cases in which it becomes necessary to do computations inside an operator that bear little resemblance to one's intuitive definition of the operator, the instances in which the author has accounted these were primarily with an older generation of compilers. It would seem that sufficient discipline could eliminate opaque uses for operators in most cases and that similarly opaque calculations could be done inside misleadingly named procedures even in the absence of defined operators.

Acknowledgments

The author gratefully acknowledges the support of the Office of Naval Research via Contract 61-8804065. The author also thanks Dr. Jeanie Osburn of the US Naval Research Laboratory for pointing out that the bisection method more closely resembles common practice in searching for bugs and Dr. Robert Armstrong of Sandia National Laboratories for many thought-provoking discussions on programming languages, operator overloading, and the limits of software estimation.

Appendix

```
module integrable_model_module
     implicit none
     private
     public :: integrate

     type ,abstract ,public :: integrable_model
     contains
       procedure(time_derivative      ) ,deferred :: d_dt
       procedure( symmetric_operator  ) ,deferred :: add
       procedure( symmetric_assignment) ,deferred :: assign
       procedure(asymmetric_operator  ) ,deferred :: multiply
       generic  :: operator(+)   => add
       generic  :: operator(*)   => multiply
       generic  :: assignment(=) => assign
     end type integrable_model
     abstract interface
       function time_derivative(this) result(dState_dt)
         import :: integrable_model
         class(integrable_model) ,intent(in)  :: this
         class(integrable_model) ,allocatable :: dState_dt
       end function time_derivative
       function symmetric_operator(lhs,rhs) result(op_result)
         import :: integrable_model
         class(integrable_model) ,intent(in)  :: lhs,rhs
         class(integrable_model) ,allocatable :: op_result
```

```
    end function symmetric_operator
    function asymmetric_operator(lhs,rhs) result(op_result)
      import :: integrable_model
      class(integrable_model) ,intent(in)  :: lhs
      class(integrable_model) ,allocatable :: op_result
      real                    ,intent(in)  :: rhs
    end function asymmetric_operator
    subroutine symmetric_assignment(lhs,rhs)
      import :: integrable_model
      class(integrable_model) ,intent(in)    :: rhs
      class(integrable_model) ,intent(inout) :: lhs
    end subroutine symmetric_assignment
  end interface
contains
  subroutine integrate(integrand,dt)
    class(integrable_model) :: integrand
    real ,intent(in)        :: dt
    integrand = integrand + integrand%d_dt*dt
  end subroutine
end module integrable_model_module
```

References

[1] G.J. Chaitin, How to run algorithmic complexity theory on a computer: Studying the limits of mathematical reasoning, *Complexity* **2**(1) (1996), 15–21.

[2] N.E. Fenton and N. Ohlsson, Quantitative analysis of faults and failures in a complex software system, *IEEE Trans. Soft. Eng.* **26** (2000), 797–814.

[3] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd edn, IEEE Computer Society, Los Alamitos, CA, 1997.

[4] G. Golub and J. Ortega, *Scientific Computing and Differential Equations: An Introduction to Numerical Methods*, Academic Press, 1991.

[5] P.W. Grant, M. Haveraaen and M.F. Webster, Coordinate free programming of computational fluid dynamics problems, *Sci. Program.* **8** (2000), 211–230.

[6] L. Hatton, The T Experiments: Errors in scientific software, *IEEE Comput. Sci. Eng.* **4** (1997), 27–38.

[7] W.D. Henshaw, Overture: An object-oriented framework for overlapping grid applications, UCRL-JC-147889, Lawrence Livermore National Laboratory, 2002 (also in: *32nd AIAA Conference on Applied Aerodynamics*, St. Louis, MO, June 2002, pp. 24–27).

[8] S.R. Kirk and S. Jenkins, Information theory-based software metrics and obfuscation, *J. Sys. Soft.* **72**(2) (2004), 179–186.

[9] J.P. Lewis, Limits to software estimation, *ACM SIGSOFT Soft. Eng. Notes* **26**(4) (2001), 54–59.

[10] K. Long, Sundance 2.0 Tutorial, Sandia National Laboratories Technical Report SAND2004-4793, 2004.

[11] J.D. Musa, A. Iannino and K. Okumoto, *Engineering and Managing Software with Reliability Measures*, McGraw-Hill, 1987.

[12] S. Oliveira and D. Stewart, *Writing Scientific Software: A Guide to Good Style*, Cambridge University Press, 2006.

[13] R. Pressman, *Software Engineering: A Practitioner's Approach*, 5th edn, McGraw-Hill, 2001.

[14] D.W.I. Rouson, R. Rosenberg, X. Xu, I. Moulitsas and S.C. Kassinos, A grid-free abstraction of the Navier–Stokes equations in Fortran 95/2003, *ACM Trans. Math. Soft.* **34**(1), 2008.

[15] D.W.I. Rouson, H. Adalsteinsson and J. Xia, Design patterns for multiphysics modeling in Fortran 2003 and C++, *ACM Trans. Math. Soft.* (2008), in review.

[16] D.W.I. Rouson, X. Xu and K. Morris, Formal constraints on memory management in composite overloaded operations, *Scientific Programming* **14** (2006), 27–40.

[17] S. Schach, *Object-Oriented and Classical Software Engineering*, 5th edn, McGraw-Hill, New York, 2002.

[18] A. Shalloway and J.R. Trott, *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley, 2002.

[19] C. Shannon, A mathematical theory of communication, *Bell Syst. Techn. J.* **27** (1948), 379–423, 623–656.

[20] M. Shepperd and D.C. Ince, A critique of three metrics, *Journal of Systems and Software* **26** (1994), 197–210.

[21] W.P. Stevens, G.J. Myers and L.L. Constantine, Structured design, *IBM Systems Journal* **13**(2) (1974), 115.