Editors: Konrad Hinsen, konrad.hinsen@cnrs-orleans.fr | Matthew Turk, matthewturk@gmail.com

Writing Software Specifications

Konrad Hinsen | Centre de Biophysique Moléculaire in Orléans

A question that computational scientists should constantly worry about is whether their software actually computes what they think it computes. After all, computer programs are complex artifacts and thus likely to contain mistakes. This becomes painfully evident when a program crashes or produces nonsense output, but even an apparently credible result could well be wrong. The most widely used technique today to improve the correctness of computations is *testing*, a subject that comes up frequently in the Scientific Programming department.[1] A better approach, at least in principle, is formal validation, which involves either a mathematical proof of correctness, or an exhaustive validation for all possible program inputs. Such techniques are under active development, but for now applied only to small and/or safety-critical software projects because of the heroic efforts they demand.[2] However, this is likely to change in the near future.

No matter how you go about checking that your program does what you think it does, a necessary first step is to write down precisely what the program should do, and what error tolerance, if any, you're willing to accept. This is called a *specification*. In scientific computing, a specification most often takes the form of a verbal description plus some mathematical equations—that is, it uses the same notation that we use in scientific articles to explain computational methods to our peers. For simple specifications, this works quite well. However, as computational methods are applied to ever more complex systems, the limits of such an informal notation become apparent. One problem is that it's difficult to be certain that an informal specification is complete and free of contradictions. Another problem is that it can't be processed by computer programs. Testing against an informal specification requires a manual translation of the specification into test cases, another step where mistakes can easily creep in. A *formal specification*, one that's in a computer-readable language, is a first step to solving these problems, but we should never expect miracles: to err is human, and no formalism or automated procedure will ever guarantee the absence of mistakes.

In the following, I motivate the need for formal specifications using an example from my own field of research, biomolecular simulation. However, I'm sure that readers with different backgrounds will recognize the fundamental problem of specifications that are too complex to work with on paper. A specification language called Maude can be used to create formal specifications for such situations.[3] I admit from the start that neither Maude nor to my knowledge any other existing specification language is adequate for the task of specifying complex scientific computations. The aim of this article is not to present a ready-to-use technique but to prepare you for a long-term evolution in computational science that is in my opinion inevitable.

An Illustration from Biomolecular Simulation

The most widely used computational model for biological macromolecules (proteins, DNA, and so on) is known as the molecular mechanics model. It treats each atom as a point mass obeying the laws of classical mechanics (Newton's equations of motion). The interactions in a system of point masses are described by a potential energy function, which assigns a number (the potential energy) to any specific configuration of the atoms. In molecular simulation, that potential energy function is called a *force field*. The parameters of a force field must be fit to a combination of experimental data and results of computations at a more fundamental level, which is quantum chemistry. This is a significant effort, and therefore there are only a few widely used biomolecular force fields. The one I use here for illustration is called Amber (Assisted Model Building with Energy Refinement; http://ambermd.org). In a typical scientific article describing a simulation-based study, the Amber force field is summarized by a formula such as

$$U = \sum_{\text{bonds } ij} k_{ij}\left(r_{ij} - r_{ij}^{(0)}\right)^2 + \sum_{\text{angle } ijk} k_{ijk}\left(\phi_{ijk} - \phi_{ijk}^{(0)}\right)^2 + \sum_{\text{dihedrals } ijkl} k_{ijkl}\cos\left(n_{ijkl}\theta_{ijkl} - \delta_{ijkl}\right) + \sum_{\text{all pairs } ij} 4\varepsilon_{ij}\left(\frac{\sigma_{ij}^{12}}{r^{12}} - \frac{\sigma_{ij}^{6}}{r^{6}}\right) + \sum_{\text{all pairs } ij} \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}}.$$

This conveys the important information that the potential energy is the sum of five distinct terms and describes the general form of these terms. But it leaves many open questions. To compute a sum over all bonds, we need to know where exactly these bonds are and how each bond's parameters, the force constant k_{ij} and the equilibrium length $r_{ij}^{(0)}$, are obtained. Moreover, the above formula contains white lies and omissions. The last two terms aren't really sums over all pairs of atoms because some pairs must be excluded, and the functional forms of the last two terms are in practice always approximated for efficiency reasons.

While most of these aspects are discussed somewhere in the scientific literature, no comprehensive specification says "*this* is the Amber force field." In fact, some details of the potential energy computation aren't documented anywhere else than in the actual source code of a program that performs the computation. And because there are several such programs, each of them actually implements its own variant, meaning that the Amber force field isn't a precisely specified model, but rather a family of models. However, many practitioners of biomolecular simulation aren't even aware of this fact. Unless you try to write your own implementation, you might never realize that the specification in the scientific literature is incomplete.

You might be tempted to say that a scientific model as complex as the Amber force field should be specified by the program that computes it and accept that each scientific article only describes selected aspects of this model. However, this doesn't work for several reasons. First, more than one program claims to implement the Amber force field. The scientific community would have to declare one of them to be the authoritative implementation. And because program source code changes in the course of ongoing maintenance, the community would have to choose a specific version as well. A second problem is that a program's source code isn't a precise specification of its results. The results depend also on the compiler being used and on the computer the program is run on. This is particularly true for programs using floating-point operations, which are subject to compiler-specific optimizations that can lead to changes in the results. Finally, on a more fundamental level, the whole point of a specification is to have something to test programs against. Defining the program as its own specification makes any testing impossible.

Formal Specifications

The above example illustrates that a formal specification must be somewhere in between an informal specification and a piece of software. Like an informal specification, it must limit itself to defining the results of a computation, leaving aside technical details such as performance, memory management, I/O, and variations in computational platforms. Like software source code, it must be amenable to automated processing. This includes execution—that is, computing concrete results for concrete inputs—as well as automatic code generation, for creating test cases and automated proofs. Such a specification must therefore be expressed in a formal language, with well-defined syntax and semantics.

There are several fundamental approaches to specification languages that I won't discuss here. The language Maude, which I use for illustration, belongs to the category of *algebraic specification languages*, more specifically to the OBJ family of languages derived from the language OBJ, which was published in 1976. The theoretical foundation of the OBJ family is provided by *equational logic* and *term rewriting*. Equational logic is a formal system of reasoning whose rules are based on the principle that replacing equals by equals in a true statement yields another true statement. Term rewriting is a computational paradigm based on modifying expressions by applying transformation rules. It's widely used in computer algebra systems.

A term rewriting system is built on a single data structure, called (not surprisingly) a *term*, which is very similar to what's called a term in mathematics. Formally, a term is defined as any expression of the form $op(arg_1, \ldots, arg_N)$, where op is an *operator*, and the arguments arg_1 to arg_N are terms. Each operator has a fixed number of arguments, and the special case of a zero-argument operator is used to represent constant values. The specification of a term algebra includes a list of the allowed operators and the number of arguments required for each one. Maude uses a variant called *order-sorted* term algebra, in which each term also has a declared *sort*, which is what many programming languages call a *type*. Moreover, sorts can be declared to be subsorts of other sorts, creating a partial order on the set of all sorts. This resembles inheritance in object-oriented languages but is much more flexible.

As a first example, Figure 1 shows a basic Maude definition for a Boolean algebra. The stars around the operator names are there to keep them distinct from Maude's built-in operators for Boolean logic. This piece of code defines a

```
fmod BOOLEAN is

  sort Boolean.

  op *true* : -> Boolean.
  op *false* : -> Boolean.

  op *not* : Boolean -> Boolean.
  op *and* : Boolean Boolean -> Boolean.
  op *or* : Boolean Boolean -> Boolean.

  var A : Boolean.

  eq *not* (*true*) = *false*.
  eq *not* (*false*) = *true*.

  eq *and* (*true*, A) = A.
  eq *and* (A, *true*) = A.
  eq *and* (*false*, A) = *false*.
  eq *and* (A, *false*) = *false*.

  eq *or* (*true*, A) = *true*.
  eq *or* (A, *true*) = *true*.
  eq *or* (*false*, A) = A.
  eq *or* (A, *false*) = A.

endfm
```

Figure 1. A Maude module defining a simple Boolean algebra.

functional module called BOOLEAN. Maude offers another type of module, the system module, but we won't need it in this introduction. The module BOOLEAN defines a new sort Boolean, two constants of sort Boolean for true and false, and the three Boolean operators not, and, and or. The variable declaration for A says that in the following, A stands for any term of sort Boolean.

The rest of the module consists of equations, which in Maude have an interesting double interpretation. First, they state that two terms, or term patterns containing variables, are mathematically equal. Second, each equation can be used as a simplification rule. When asked to *reduce* a term, which is term rewriting jargon for "simplify as much as possible," Maude checks if any subterm matches the left-hand side of an equation, and if it does, replaces it by the right-hand side. It goes on doing such replacements until there's no more subterm that matches any left-hand side of an equation. Such a nonreducible term is called a *normal form*. It isn't evident that a given set of equations leads to a unique normal form for each possible term, and in fact this doesn't hold in general. An in-depth discussion of this and related issues appears elsewhere.[4]

If you put the above definition into a file called boolean.maude, you can then start a Maude session and type

```
load "boolean.maude".
```

being careful not to forget the space before the period at the end. You can then ask Maude to evaluate terms, that is,

```
reduce *and*(*not*(*true*), *or*(*false*,
  *not*(*false*))).
```

which yields *false* as a result. For your convenience, the file boolean.maude and the other examples from this article are available at http://github.com/khinsen/cise-software-specifications.

From these examples it becomes clear that terms do double duty as data structures and code. A function or procedure in a traditional programming language is the equivalent of an operator with arguments and equations, such as our *not* or *and*. Zero-argument operators represent constants, such as our *true*. However, this analogy isn't complete because a term rewriting system admits terms that *could* be rewritten but aren't. For example, if I remove the equation *not*(*false*) = *true* from the module BOOLEAN, then any subterm *not*(*false*) will simply remain as it is, whereas *not*(*true*) is replaced by *false*. In fact, term rewriting is more similar to algebraic manipulations done on mathematical formula than to function evaluation in programming languages. Reduction to normal form is exactly what a computer algebra system calls simplification of a formula: if there's a simplification rule, it's applied, otherwise a term remains as it is.

This is almost all you need to know about Maude for now. Maude offers a number of convenience features that make writing specifications a lot simpler. For example, terms can be written with different syntax than *op*(arg_1, …, *argN*), allowing for better readability in complex expressions. Maude also allows numbers (natural, integer, rational, and floating point) as terms and provides a more concise notation for associative and commutative operators, which avoids having to write equations for both *and*(*true*, A) and *and*(A, *true*) in the above example. I'll use such features sparingly to keep the examples easy to follow.

A Specification for an Atomic Simulation

The full Amber force field discussed earlier is much too big to serve as an example. I therefore limit myself to a single term, the next-to-last one, known as a Lennard-Jones potential. On its own, it describes the noble gases rather well, which is why I use a simulation of atomic argon as an example. Our potential energy then is

$$U = \sum_{\text{all pairs } ij} 4\varepsilon\left(\frac{\sigma^{12}}{r_{ij}^{12}} - \frac{\sigma^6}{r_{ij}^6}\right), \qquad (1)$$

```
from math import sqrt

class Vector(object):

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

class Cell(object):

    def check_configuration (self, configuration):
        raise NotImplementedError

    def distance (self, p1, p2):
        raise NotImplementedError

class InfiniteCell(object):

    def check_configuration(self, configuration):
        pass

    def distance (self, p1, p2):
        return sqrt (( p2.x-p1.x)**2 + (p2.y-p1.y))**2 + (p2.z-p1.z))**2)

class OrthorhombicCell (object):

    def __init__(self, edges):
        self.edges = edges

    def check_configuration(self, configuration):
        for p in configuration:
            assertp.x >= 0. and p.x < self.edges[0]
            assertp.y >= 0. and p.y < self.edges[1]
            assertp.z >= 0. and p.z < self.edges[2]

    def distance (self, p1, p2):
        lx, ly, lz = self.edges
        return sqrt (self.minimum_image(p2.x-p1.x, lx)**2
                     + self.minimum_image(p2.y-p1.y, ly)**2
                     + self.minimum_image(p2.z-p1.z, lz)**2)

    def minimum_image(self, d, l):
        if d > 0.5*l:
            d -= l
        elif d < -0.5*l:
            d +=l
        return d

def pair_energy(r):
    LJEnergy = 1.    # kJ/mol
    LJRadius = 0.34  # nm
    LJCutoff  = 1.5  # nm
    sr6 = LJRadius **6 / r**6 if (r < LJCutoff ) else 0
    return 4 * LJEnergy * (sr6 * sr6 - sr6)

def potential_energy(cell, configuration):
    cell.check_configuration(configuration)
    n = len(configuration)
    e = 0.
    for i in range(n):
        for j in range(i+1, n):
            r = cell.distance(configuration [i],
                              configuration [j])
            pe = pair_energy(r)
            e += pe
    return e

# Two test cases : a triangle and a cubic lattice

def triangle(h):
    cell = InfiniteCell()
    configuration = [Vector(0, 0, 0),
                     Vector(h, 0, 0),
                     Vector(0.5*h, 0.5*sqrt (3)*h, 0)]
    return potential_energy(cell, configuration)

def cubic_lattice(n, h):
    cell = OrthorhombicCell(np.array ([n*h, n*h, n*h]))
    configuration = [Vector(h*x, h*y, h*z)
                     for x in range(n)
                     for y in range(n)
                     for z in range(n)]
    return potential_energy(cell, configuration)


print triangle (0.3)
print cubic_lattice (2, 0.375)
print cubic_lattice (4, 0.375)
```

Figure 2. A Python program implementing the informal specification for the argon potential energy.

where for argon the parameters are $\varepsilon = 1$ kJ/mol and $\sigma = 0.34$ nm. For efficiency, the potential energy is set to zero, and thus not computed at all, for pairs whose r_{ij} is larger than a cutoff value r_c, which I choose to be 1.5 nm.

One more aspect needs to be addressed. Simulations in material science are usually done with *periodic boundary conditions*, which means that you simulate a small box (I use a cubic one) filled with atoms or molecules, which you then imagine to be surrounded by an infinite number of copies of itself on a lattice. The goal is to have a system without surfaces and thus eliminate surface effects that, for the size of the system you can actually afford to simulate, would be much bigger than in real-life situations. Another way to describe this construction is as replacing each atom with an infinite cubic lattice of identical atoms that always move together. This raises the question of how the "sum over all pairs" is defined. The most common convention, which yields an exact result for systems whose edge length is larger than 2 r_c, is to consider only pairs of atoms inside a single image of the cubic box but define the distance between them as the shortest possible distance between any image of atom i and any image of atom j. This is called the minimum-image convention.

The two preceding paragraphs are an informal specification for the potential energy in our argon simulation. Please convince yourself that it's complete. Imagine that you're given N positions $\mathbf{r}_i$, describing the configuration of N argon atoms in a cubic simulation box. You also get the edge length l of the box. Can you compute the potential energy with this information? Can you write a computer program for doing this computation?

If you answered yes to both questions, you've probably fallen into my well-prepared trap, but let's continue. Figure 2 shows a simple Python program that implements the above specification. It has intentionally not been optimized in any way. For ease of use in testing, it provides both the periodic geometry discussed earlier, to be used for liquids and gases, and standard "infinite box" geometry, which is suitable for atom clusters. Even if what you really want to simulate is periodic systems, it helps to have the simpler infinite geometry available as well because it makes for simpler test cases. If a "periodic" test case fails but the "infinite" ones pass, you know that your bug is probably in the application of the minimum-image convention.

If you answered yes to my two questions and the program you had in mind resembles the one in Figure 2, then you've definitely fallen into my trap. This program uses floating-point numbers wherever the informal specification uses...well, probably real numbers, though it could also be rational numbers. It doesn't really matter because the potential energy equation is equally valid for both. For floating-point numbers, the equation is insufficient because the order of all operations must be specified. In particular, a specific order of summation over the atom pairs must be chosen. For bigger systems, different summation orders do lead to visibly different results, so this isn't just an academic exercise. Moreover, round-off errors in floating-point computations mean that the resulting potential energy isn't the same as the correct one computed using rational numbers. It's an approximation, and approximations need to be spelled out explicitly in a specification.

If you answered yes to my two questions and planned to use rational arithmetic in your program, you may now pat yourself on the back for having successfully avoided the floating-point trap. However, you should also realize that you've been very lucky: of the five terms in the Amber force field, I chose the only one for my example that can in fact be computed using rational arithmetic because it contains no square roots or transcendental functions. To be precise, the Python code and the upcoming Maude specification *do* use square roots to be as close as possible to the informal specification, but the potential energy could also be written in terms of r^2.

A formal specification in Maude for the argon potential energy is shown in Figure 3 and the test cases in Figure 4. First, let's look at some additional Maude features used in this code.

A much-used Maude feature is the inclusion of a previously defined module into another module, indicated by the keyword `including`. This permits decomposing a specification into small units that are easy to understand and reuse.

In addition to functional modules (`fmod`), you see functional theories (`fth`) and views (`view`). Theories take their name not from scientific theories but from theories in mathematical logic. Their closest analogs in standard programming languages are interface specifications. A Maude theory defines sorts, operators, and equations that a complying module must define as well and that client modules can rely on. The mapping from an interface to an implementation is very flexible in Maude but also rather verbose: it's necessary to define a view that states which sorts and operators in a module correspond to which sorts and operators in a theory. The code in Figure 3 defines one theory, `CELL`, which corresponds to the abstract base class in the Python version. It also uses a predefined theory, `TRIV`, which defines the criteria that elements of a list must satisfy: none. In fact, the theory `TRIV` contains nothing but a single sort, `Elt`, with no equations attached. The use of `TRIV` means, "if you want a list whose elements are of sort X, you must define sort X." In a programming language, such pedantry would be considered cumbersome, but in a specification language, precision takes priority over convenience, and parsimony in the language specification is more important than expressivity.

In the test cases, I've chosen to write out explicitly the configurations for the cubic lattices. It's possible to compute them in Maude in much the same way as in the Python version, but this would have required introducing additional Maude features.

With those explanations out of the way, we can now discuss the important differences between the Python code in Figure 2 and the Maude specification in Figure 3. Their overall structure is similar, so you might well believe you're looking at the same program written in two different languages. This isn't completely wrong, as the difference between specification languages and programming languages isn't a fundamental one but a question of different priorities. The key difference is that the Maude code defines *equations*, whereas the Python code contains *statements*, that is, instructions for the computer to carry out. This difference is somewhat obscured by the fact that Maude equations do double duty as simplification rules, which resemble statements. But equations are more versatile than statements: equations can be put to other uses, such as deductive proofs or code generation. There are in fact a few Maude tools available that take modules as input and perform certain kinds of proofs on their contents, but I won't discuss them here.

An immediate consequence of the lack of statements in Maude is that there are no control structures in the standard sense, and in particular, no loops. Readers familiar with functional programming will see that this isn't a problem, as loops can be replaced by recursive function calls. There are no recursive function calls either in Maude because there are no functions. Recursion is expressed just like in mathematics: as an equation involving the same operator applied to simpler arguments. This is how the loop over atom pairs is defined in the last equation of module `LENNARD-JONES-ENERGY`.

Finally, it's worth noting that in the Maude specification, floating-point operations have a precisely defined order, which isn't true in the majority of programming languages. Python also happens to honor the order of operations written down by the programmer but doesn't promise to do so. It's simply how the current implementation happens

```
fmod  VECTOR  is
    including FLOAT.
    sort  Vector.
    op  v: Float Float Float -> Vector.
endfm

view  Vector from TRIV to VECTOR is
  sort  Elt to Vector.
endv

fmod  CONFIGURATION  is
    including  VECTOR.
    including  LIST{Vector} * (sort List{Vector} to Configuration).
endfm

fth CELL is
    including VECTOR.
    sort  Cell.
    op  distance : Cell Vector Vector -> Float.
endfth

  fmod  INFINITE-CELL is
    including  FLOAT.
    including  VECTOR.
    sort  Cell.
    op  universe : -> Cell.
    op  distance : Cell Vector Vector -> Float.
    vars  x1 y1 z1 x2 y2 z2 : Float.
    var  u : Cell.
    eq  distance(u, v(x1, y1 ,z1), v (x2, y2, z2)) =
          sqrt (((x2 - x1) ^ 2.0) + ((y2 - y1) ^ 2.0) + ((z2 - z1) ^ 2.0)).
endfm

view  INFINITE-CELL from CELL to INFINITE-CELL is
endv

fmod  ORTHORHOMBIC-CELL is
    including FLOAT.
    including VECTOR.
    sort Cell .
    op  universe : Float Float Float -> Cell.
    op  distance : Cell VectorV ector -> Float.
    vars  lx ly lz x1 y1 z1 x2 y2 z2 dx : Float.
    eq  distance(universe(lx, ly, lz), v(x1, y1, z1), v(x2, y2, z2)) =
          sqrt((d(lx, x2 - x1) ^ 2.0)
             + (d(ly, y2 - y1) ^ 2.0)
             + (d(lz, z2 - z1) ^ 2.0)).
    op  d : Float Float -> Float.
    eq  d(lx, dx) = if dx > (lx / 2.0)
           then dx  - lx
           else if dx < -(lx / 2.0)
              then dx + lx
              else dx fi fi.
endfm

view ORTHORHOMBIC-CELL from CELL to ORTHORHOMBIC-CELL is
endv

fmod  LENNARD-JONES-PAIR is
    including FLOAT.
    op  LJRadius : -> Float.
    eq  LJRadius = 0.34 . *** nm
    op  LJEnergy : -> Float.
    eq  LJEnergy = 1.0 .  ***  kJ/mol
    op  LJCutoff : -> Float.
    eq  LJCutoff = 1.5 . *** nm
    var  R : Float.
    op  $sr6 : Float -> Float.
    eq  $sr6(R) = if (R < LJCutoff)
                   then (LJRadius ^ 6.0) / (R ^ 6.0)
                   else 0.0 fi .
    op  pairEnergy : Float -> Float.
    eq  pairEnergy(R) = 4.0 *LJEnergy *($sr6 (R) *$sr6 (R) - $sr6(R)).
endfm

fmod LENNARD-JONES-ENERGY{U :: CELL} is
    including FLOAT.
    including LENNARD-JONES-PAIR.
    including VECTOR.
    including CONFIGURATION.
    op potentialEnergy : U$Cell Configuration -> Float.
    var U : U$Cell.
    var  R : Configuration.
    vars  R1 R2: Vector.
    eq  potentialEnergy(U, R1) = 0.0.
    eq  potentialEnergy(U,R1R) = oneWithOthers (U,R1,R)
                                 + potentialEnergy (U,R).

    op oneWithOthers : U$Cell Vector Configuration -> Float.
    eq  oneWithOthers(U, R1, R2) = pairEnergy(distance(U, R1, R2)).
    eq  oneWithOthers(U, R1, R2R) = pairEnergy(distance(U, R1, R2))
                                    + oneWithOthers(U, R1, R).
endfm
```

Figure 3. A Maude specification for the argon potential energy.

to work. In contrast, the arithmetic operators defined in Maude's standard library specify associativity for integers and rational numbers but not for floating-point numbers. An attempt to prove equality for two expressions that differ only in the order of operations would succeed for the former but fail for the latter.

```
fmod TRIANGLE is
    including  INFINITE-CELL.
    including  LENNARD-JONES-ENERGY{INFINITE-CELL}.
    including  CONFIGURATION.
    op  h : -> Float.
    eq  h = 0.3.
    op  conf : -> Configuration.
    eq  conf = v(0.0, 0.0, 0.0) v(h, 0.0, 0.0)
             v(0.5 *h, 0.5 *h *sqrt (3.0), 0.0).
endfm

fmod CUBIC-LATTICE is
    including ORTHORHOMBIC-CELL.
    including LENNARD-JONES-ENERGY{ORTHORHOMBIC-CELL}.
    including CONFIGURATION.
    including INT.
    including CONVERSION.
    op u : -> Cell.
    op conf : -> Configuration.
    op h : -> Float.
    eq h = 0.375.
    vars XYZ : Int.
    op point : IntIntInt -> Vector.
    eq point(X, Y, Z) = v(h *float (X), h *float (Y), h *float (Z)).
endfm

```

```
fmod  CUBIC-LATTICE-2 is
   including  CUBIC-LATTICE.
    eq  u = universe(2.0 *h, 2.0 *h, 2.0 *h).
    eq  conf = point(0,0,0) point(0,0,1) point(0,1,0) point(0,1,1)
               point(1,0,0) point(1,0,1) point(1,1,0) point(1,1,1).
endfm

fmod CUBIC-LATTICE-4 is
   including  CUBIC-LATTICE.
    eq  u = universe(4.0 *h, 4.0 *h, 4.0 *h).
    eq  conf = point(0,0,0)point(0,0,1) point(0,0,2)point(0,0,3)
               point(0,1,0) point(0,1,1) point(0,1,2) point(0,1,3)
               point(0,2,0) point(0,2,1) point(0,2,2) point(0,2,3)
               point(0,3,0) point(0,3,1) point(0,3,2) point(0,3,3)
               point(1,0,0) point(1,0,1) point(1,0,2) point(1,0,3)
               point(1,1,0) point(1,1,1) point(1,1,2) point(1,1,3)
               point(1,2,0) point(1,2,1) point(1,2,2) point(1,2,3)
               point(1,3,0) point(1,3,1) point(1,3,2) point(1,3,3)
               point(2,0,0) point(2,0,1) point(2,0,2) point(2,0,3)
               point(2,1,0) point(2,1,1) point(2,1,2) point(2,1,3)
               point(2,2,0) point(2,2,1) point(2,2,2) point(2,2,3)
               point(2,3,0) point(2,3,1) point(2,3,2) point(2,3,3)
               point(3,0,0) point(3,0,1) point(3,0,2) point(3,0,3)
               point(3,1,0) point(3,1,1) point(3,1,2) point(3,1,3)
               point(3,2,0) point(3,2,1) point(3,2,2) point(3,2,3)
               point(3,3,0) point(3,3,1) point(3,3,2) point(3,3,3).
endfm

reduce in TRIANGLE : potentialEnergy(universe, conf).
reduce in CUBIC-LATTICE-2 : potentialEnergy(u, conf).
reduce in CUBIC-LATTICE-4 : potentialEnergy(u, conf).
```

Figure 4. Maude test cases for the argon potential energy.

Specifications in Real Life

While the above example illustrates nicely how a specification language works and how it differs from a programming language, any attempt to use Maude or any similar language for complex scientific models soon shows their limits. The biggest problem with Maude is that it works in complete isolation from other computational tools. You can't even read files from Maude. The only input to Maude is a sequence of Maude commands. All data you want to work on in Maude must exist in the form of valid Maude modules. Imagine, for example, a specification for the full Amber force field. It must contain the few hundreds of numerical parameters that all molecular simulation programs read from a file. This information would have to be converted to a Maude module, which is an error-prone process. It thus isn't evident that Maude would use the same values as numerical implementations of the force field. You could in principle require that all numerical implementations read their parameters from Maude modules, but implementing a parser for Maude modules is a nontrivial task. And numerical parameters are only the simplest part of the data that make up the Amber force field: there's also a database of molecular fragments with associated parameters. While Maude makes it straightforward in principle to write code that analyzes and transforms Maude modules, in practice, such code has to be written in Maude.

There is, of course, a reason why Maude accepts data only in its own language. As I explained earlier, the mathematical theories underlying Maude rely on a very simple data model: there's nothing but terms and equations, leaving no room for files or for different basic data types, such as arrays. Introducing any of these would reduce the fundamental simplicity that facilitates mathematical reasoning about Maude modules. However, this doesn't mean that a better integration of specification and programming languages is impossible—it's simply an aspect that hasn't yet been explored well enough. For example, one solution that looks feasible is to provide "data adaptors" that read in data in various formats and present it to Maude as terms. But the problem of language isolation is more general; what computational scientists really need is an integrated system for writing specifications *and* implementations at different levels of optimization, with transitions between these levels made as painless as possible. I've written about such a system before,[5] but it remains a dream.

Ultimately, there's a chicken-and-egg problem: developers of specification languages won't work on scientific applications unless there's a clear demand from the scientific computing community, but computational scientists won't get interested in specification languages until they see a clear utility in them for their daily work. I hope this article contributes a bit to the establishment of closer contacts between these two communities.

In the meantime, what can scientific software developers do to have something better than informal specifications? One option is a simple reference implementation of the kind shown in Figure 2. It's both easier to understand and to debug than an optimized implementation for production use, and it can be put to good use in a test suite. Using a programming language for writing specifications requires a radical change of attitude: criteria such as simplicity, clarity, and minimal dependencies take priority over efficiency and modularity. If you're an experienced Python programmer and your first reaction to Figure 2 was, "I'd use NumPy here," you've shown good habits for an application or library developer but also bad habits for a specification author. In fact, NumPy is a nontrivial dependency whose use adds nothing in terms of precision, simplicity, or clarity.

Writing a simple reference implementation is also a good way to "test" an informal specification, which is what will get published in the scientific literature. To make best use of this, the reference implementation should be written by someone else than the informal specification. If the implementer requires any additional information or must make unstated assumptions, then the informal specification needs a revision.

Like other quality assurance measures, writing specifications might initially seem to be too much of an effort, taking time and resources that could be better spent "doing science." However, ensuring the correctness of computations *is* part of "doing science." My personal experience is that specifications pay off as early as in the debugging phase of nontrivial scientific software. So, please, give it a try. ■

References

1. P.F. Dubois, "Testing Scientific Programs," *Computing in Science and Eng.*, vol. 14, no. 4, 2012, pp. 69–73.
2. S. Boldo et al., "Trusting Computations: A Mechanized Proof from Partial Differential Equations to Actual Program," *Computers & Mathematics with Applications*, vol. 68, no. 3, 2014, pp. 325–352.
3. M. Clavel et al., "All about Maude: A High-Performance Logical Framework," *LNCS 4350*, Springer, 2007; www.springer.com/computer/swe/book/978-3-540-71940-3.
4. F. Baader and T. Nipkow, *Term Rewriting and All That*, Cambridge Univ. Press, 1998.
5. K. Hinsen, "Daydreaming about Scientific Programming," *Computing in Science and Eng.*, vol. 15, no. 5, 2013, pp. 77–79.

Konrad Hinsen is a researcher at the Centre de Biophysique Moléculaire in Orléans (France) and at the Synchrotron Soleil in Saint Aubin (France). His research interests include protein structure and dynamics and scientific computing. Hinsen has a PhD in theoretical physics from RWTH Aachen University (Germany). Contact him at konrad.hinsen@cnrs-orleans.fr.