Dummy chapter

1

Reproducible Physical Science and the Declaratron

Peter Murray-Rust
pm286@cam.ac.uk
Department of Chemistry
University of Cambridge

Dave Murray-Rust
d.murray-rust@ed.ac.uk
Department of Informatics
University of Edinburgh

CONTENTS

1.1 Introduction

In this chapter we address reproducibilitiy in the Physical Sciences (PhysSci). We focus on disciplines concerned with chemistry, crystallography and materials science, although the strategies, and the Declaratron software we describe, have much greater applicability. While some scientific experiments are difficult or impossible to reproduce due to factors such as resource constraints (the Large Hadron Collider [2]) or reliance on rare occurrences (the Schoemaker-Levy Jupiter impact [15]), there is a long tradition of reproducibility in PhysSci. This is based on the almost Platonic identity of materials: to a first practical approximation, sodium chloride crystals have the same properties wherever and however they are produced.

Over the last 50 years it has become possible to measure, and now calculate, the properties of substances to a high degree of consistency. Substances are key to both research and technological exploitation (semiconductors, optical materials, piezoelectricity, second-harmonic generation and much more). We estimate that over 1 billion USD is spent on the computation of materials properties (c.f. the Material Genomics Project [16]). There is a dynamic interplay between the measurement of *observables*, or properties which are discovered through measurement of physical substances, and *computables*, which properties that can be computed through computation and simulation of physical laws. As knowledge and techniques are improved, sometimes one is more accessible or accurate, and sometimes the other, as observations feed back into more accurate computation and vice versa. There is also a fundamental reliance on instrumentation as a mechanism for reproducibility: given instruments of similar quality, there is an expectation that if an experiment is performed again, the results will be *compatible* with the original results.

The following instrumental techniques have now developed both automation (e.g. robot sample feeders) and high-throughput (hundreds of samples per day or more): Single-crystal X-ray crystallography, powder X-ray diffraction, IR/UV/VIS, [9], Nuclear Magnetic Resonance spectroscopy [4], Mass spectrometry [8], and many more). There are a few standards (e.g. JCAMP-DX [1] and AniML [18] [2]) but most output is proprietary so there are very few effective communal dictionaries and semantics.

As well as self-consistency of results, reproducibility increasingly requires agreement between experiment and theory. Computational experiments, including calculation and simulation, are part of the architecture of an agreement between theory and experiment, through the process of turning models of reality into numbers which can be compared to observed results. This implies that there is a need for mechanisms and expectations of reproducibility to be applied equally to scientific computation as instrumentation. In this chapter we describe how existing technologies can be combined with a novel approach to semantic calculation to carry out reproducible scientific computation. To demonstrate this, we take as a case study the task of reliably computing the properties of a material given a completely well defined semantic specification.

1.1.1 What do we mean by reproducible computation?

1.1.1.1 An archetypal example of the problem

There are two main methods of computing the properties of matter. Quantum Mechanics (QM) involves solving Schroedinger's equation for a multinucleus-multi-electron system.

[1] `http://www.jcamp-dx.org/`
[2] `http://animl.sourceforge.net/`

$$
\begin{aligned}
V(\mathbf{r}) \quad = \quad & \sum_{bonds} K_b(b-b_0)^2 + \sum_{angles} K_\theta(\theta-\theta_o)^2 \\
& + \sum_{dihedrals} (V_n/2)(1+\cos[n\phi-\delta] \\
& + \sum_{nonbij} (A_{ij}/r_{ij}^{12}) - (B_{ij}/r_{ij}^{6}) + (q_i q_j / r_{ij})
\end{aligned}
$$

FIGURE 1.1
Functional form of the AMBER forcefield

There is no analytical solution and the approximations can be very expensive often rising with N^3 or greater. Accuracy requires additional expense. The cheaper alternative is "Forcefields" (FF) with empirical parameterisation of Newton's laws and this forms the main example here.

In the FF approach the energy of a molecule can be approximated by a number of empirical terms together making up a "forcefield". A typical and widely used example is AMBER which contains 5 terms (see Figure). The molecule is described (emprically) by bonds, angles, torsions (dihedrals), non-bonded ("bumping atoms") and electrostatic . AMBER (the example here) will compute the energy as the sum over all components. A typical protein molecule might have several thousand bonds and angles and even more non-bonded interactions.
[3]

The point here is that it is formally impossible to relate the equation given to the actual operation of the program. The enumeration of non-bonded terms, for example, is hidden deep in the FORTRAN and may change during the progress of a calculation. An inspection of the parameter data reveals that the sum of torsions should be a double sum (over Fourier terms as well as torsions) and the electrostatics is in c.g.s. units and is missing a factor of $4\pi\epsilon_0$. It would be impossible for anyone to recreate the program from the documentation or to formally record what had been computed.

At this point, it is necessary to discuss what we mean by reproducibility in scientific computation. When dealing with *observables*, the definition is relatively clear: by following the same experimental procedure, one should obtain the same results. However, with computation, there are alternative expectations and possibilities for what the same "experimental procedue" and "results" should mean. Taking these separately, the "same experimental procedure" could mean:

- Download the original software and data and run it.
- Download the original software, compile it for a different machine and run it with the original data.

[3]`http://ambermd.org/doc11/Amber11.pdfp.19.`. The last summation should be split into separate sums, the first with A and B terms, and the last with the electrostatics (

$$q_i q_j / r_{ij}$$

) (Coulomb inverse power law)

- Download software which carries out the same operations as originally described, and apply it to the original data.
- Read a paper, produce a new implementation of the algorithms described, and run it on the original data.
- Run any of the above programs on a refined or updated data set

And the ”same results” could mean:

- Identical output at the bit level.
- Exactly the same numbers.
- Exactly the same numbers (when run on a similar machine)
- Numbers which are within some bound of error
- Ensembles of outputs which share certain characteristics (again, within the bound of error).

This discussion of what the “same results” means is partially motivated by the chaotic nature of some physical calcuations. Many algorithms in physical sciences are completely deterministic - for example the bond length result should be identical no matter what code is used (within the bounds of floating-point imprecision). However, some algorithms contain branch points which are sensitive to precision and instabilities. The Quantum Mechanical calculation of molecular energy requires two independent optimsations - the self-consistent field (SCF) of the molecular orbitals; and the optimisation of energy against geometry to get the minimum energy structure. For many molecules these are well-constrained and the calculation proceeds essentially identically on different machines, and often with different programs that use the same basic physical model. However calculations for some systems are unstable (e.g. near a transition state in a reaction) and the behaviour is effectively unpredictable on details. Similarly, the dynamics of molecules (e.g. simulated by Newton’s laws) is inherently unpredictable. Although formally deterministic, small imprecisions cause bifurcations in trajectories which rapidly diverge. When discussing reproducibility for calculations of this type, it is problematic to talk at the level of reproducing individual runs, or exact results, and the focus must be moved to ensemble or aggregate properties. Additionally, since there is no expermental validation—e.g.trajectories of individual molecules are not usually observable—special human care is required to validate code and paremeters, as mistakes will be very difficult to detect later.

For the purpose of this paper, we take semantically defined reproducible science to be defined by:

> “Can a computational scientist (or machine) with no intrinsic domain knowledge, when given the specification, build a system which can be guaranteed to compute problems in a scientific domain and produce results which are semantically consistent, and in some sense similar[4].”

[4]We will leave open the question of exactly what “similar” means here, but specify that numbers should be approximately the same

1.1.2 What's wrong with business as usual?

Currently, computable semantics are not commonplace within scientific practice. Indeed, very few scientific domains have fully addressed computable semantics. While computational chemistry is more advanced than some areas, it is still far from i) having complete computational semantics ii) integrating the use of computational semantics into the daily lives of computational chemists. We will illustrate the problem of missing semantics using examples from widely used programs; these have been chosen as typical examples in widespread use, rather than singling out egregious offences. Many other examples—most commonly used programs—display the same issues.

This is one line of input for MOPAC, a widely used CompChem program, taken from `http://openmopac.net/manual/index.html`:

```
 H 1.092  0 120.615  1 179.979 1 10 9 11
```

In this single line, there are no *explicit* semantics at all. Taking each field (separated by groups of spaces) in turn, the *implicit* semantics are:

1. `H` is the element symbol for Hydrogen. Although this seems like a precise, commonly accepted designation, many other programs use arbitrary, non-standard abbreviations for elements, with integers or floats for nuclear charge, e.g. 'W1' 8 for oxygen in water, which could also be mistaken for tungsten
2. `1.092` is the distance in Angstrom units to 10th atom. The number `10` in field 8 is what specifies it is the 10th atom.
3. `0` is an integer flag: should this distance be allowed to vary during the computation. 0 means yes, it should be allowed to vary.
4. `120.615` is the angle in degrees between this, atom `10` and atom `9`. Again, the 10th atom is specified by the `10` in field number 8, and 9th by the `9` in field 9.
5. `1` integer flag: do not allow this angle to change
6. `179.979` is the dihedral angle in degrees between this, atom `10` atom `9` atom `11`
7. `1` integer flag: do not allow this dihedral angle to change
8. `10` means that bond length is between this atom and atom 10
9. `9` means that angle is between this atom, atom 10 and atom 9
10. `11` means that dihedral angles are between this atom and atoms 10,9,11

This is just a *typical* example, and similar issues can be found in the input specifications of many programs. This type of ad-hoc, un-marked up yet implicitly meaningful data format has enormous scope for catastrophic errors. Common causes of error include: fields are mistyped when the file is edited by hand; users have an old copy of the documentation, so column ordering or meaning can change without causing obvious errors; fields boundaries can be misplaced: is it one space between each field or any number of spaces? are tabs or spaces used as delimiters? do fields need to be justified to exact column positions? The input modules of the programs usually have no validation. 'User-friendly' GUI editors are usually program-specific and proprietary, although sometimes there is an ecology of ad hoc converters; both of these situations bring a different set of issues.

In addition to the possibility for error when managing data, a huge burden is placed on the programmers who maintain applications which read these files. They are forced to maintain parsers for poorly defined specifications, and may have to deal with different dialects of the data language as alternative interpretations come into fashion. It leads to i) brittle code with poor error handling; ii) a high barrier to entry for new programmers wanting to join projects; iii) an excessive proportion of programming effort being given over to reading.

Output is similarly problematic. This example is taken from `http://www.cup.uni-muenchen.de/ch/compchem/energy/MOPAC_output.html`:

```
        ATOM NO.  TYPE        CHARGE       ATOM  ELECTRON DENSITY
          1         O         -.3827        6.3827
          2         H          .1914         .8086
          3         H          .1914         .8086
 DIPOLE          X         Y         Z       TOTAL
 POINT-CHG.     .677      .859      .000     1.094
 HYBRID         .475      .602      .000      .767
 SUM           1.151     1.461      .000     1.860
```

This cannot be understood without being a practitioner, and/or having a manual (often out of date) and/or asking questions of humans. You must know or guess that charges are in units of *electrons* and that dipoles are in Debyes - neither are SI units. We have no idea what a `HYBRID` is or how it is calculated. It appears that `SUM = POINT-CHG + HYBRID` and so we might infer that it is probably the predicted quantity. Without complete understanding of a quantity it is by defintion irreproducible - although this particular calculation could be run again to give the same numbers, it would be impossible to construct an alternative, clean room implementation which computed the same result.

In certain cases, computations are not reproducible due to *licensing* restrictions on distribution of the *output* of proprietary programs. One major program manufacturer legally forbids the publication of complete output files; in order to have any chance of creating reproducible science, it is fundamentally necessary to publish exactly these computational details.

We believe that, in addition to inhibiting reproducibility, the issues outlined above are responsible for many millions of hours of wasted work each year, by allowing errors to go unvalidated and unnoticed, propogating through chains of experiments; through time spent understanding unclear semantics and editing brittle config files; by forcing chemists to learn to parse semi-structured text, and programmers to maintain code which has to be compatible with a fuzzy, moving target data specification.

In an example from our own experience, we autogenerated input for the GAMESS program. GAMESS has a limit of 80 characters per line (cf. Hollerith cards), and some lines exceeded this. Although the program noted this in the (voluminous) output, it did not halt, but quitely discarded the offending atom records. The result was that erroneous calculations were carried out, without a strongly visible warning. These errors were only discovered when the output was re-used in further calculations, where it caused crashes. This could have been avoided by carrying out syntactic and semantic validation in the input stage, and refusing to produce output from invalid input. In general, not much trust can be placed in legacy computational chemistry programs to carry out sufficient validation on input or output; even when such validation is carried out, it is not clear how to verify that it has happened.

1.2 Constructing Chemical Semantics

We have been inspired by the practice of Crystallography in developing a completely semantic approach to physical science. For half a century IUCr [5] and the community have insisted that crystallography is reproducible by such means as: comparing experimental data, testing programs again experimenta, and most critically the creation of a computable ontology (Crystallography Information Framework, CIF). PMR has been involved with CIF for over 20 years and has taken it as a model for Chemical Markup Language (CML) [12] which is now being adopted in computational and other chemistry. Fundamental chemistry concepts were probably solidified 80 years ago and we therefore use simpler ontologies than bioscience or HEP.

To promote awareness of the need for and value of semantics we ran two meetings at Cambridge [11], [14], [6]. These brought together a group of scientists who cared about reproducibility and interpretability through developing shared semantics (dictionaries and code). These have led to further meetings (e.g. at Pacific NorthWest Laboratory in 2011) and the determination to make key tools such as NWChem [19] and Avogadro available. We believe that if there are enough components available the world will come to see the value of semantics and gradually change over a decade.

1.2.1 A note about our software status and availability

The software described here has been developed over 2 decades with a large focus on reproducibility. The main parts (JUMBO, CML) have been distributed and are widely used, but only *implicitly* for reproducibility. JUMBOConverters(templates) provides semantic coversion for legacy files during the transition to completely reproducible computation. The Declaratron itself is novel and provides complete reproducibility. All software is in public repositories. In order to give an indication to the reader of the status of any given software component, we use the following symbols: $\oslash$ = vaporware; $\overset{\star}{-}$ = prototype (has worked for us); $\overset{\star\star}{-}$ = "alpha" (hackable by others); $\frac{\star\star}{\star}$ = usable by others; $\frac{\star\star}{\star\star}$ = in widespread use.

Our software is written in Java, using XML and XPath libraries also in Java. Other CML libraries have been written in C#, C++, Python as an Object-Oriented approach is almost essential. However many of the main third-party legacy computational programs are written in FORTRAN and are too expensive to change. To interface them into this framework, therefore requires an XML/CML wrapper; the $\frac{\star\star}{\star\star}$ library has been developed by Toby White and Andrew Walker for this purpose. It deals with a subset of the languages and concentrates on program output. [6].

1.2.2 CIF and CML as semantic languages

CIF $\frac{\star\star}{\star\star}$ [1] uses a lightweight set of primitives (item or table) with data types (char or numb) and a very extensive set of dictionaries compiled by the community and crystallographers (whether experimentalist, instrument manufacturers, or computational) use it for interchange). CIF is also used for journal submission and supports computable semantic articles.

In 1994 we [7] launched Chemical Markup Language (CML) [12] to support semantic

[5]International Union of Crystallography

[6]"Visions of a Semantic Molecular Future", and "Semantic Physical Science", sponsored by the EPSRC "Pathways to Impact" program, which supports the dissemination of research done under their auspices

[7]PMR and Henry Rzepa

chemistry. Because there are few other semantic tools in physical science we have had to create a basic (non-chemical) infrastructure, STMML (STM (Scientific Technical Medical) Markup Language) [13] [8]. This supports basic quantities, error estimations, data types and scientific units of measurement and we believe it is very widely applicable, certainly to any discipline where typed quantities with units can be understood as standalone objects. Thus temperature is not a specifically chemical quantity and we can write:

```
<html:p>It was a nice day, 21 degrees
    (<cml:scalar dataType="xsd:double" dictRef="iupac:T06321"
        min="19.1" max="23.1" units="nist:sp811.08.8.5"/>)</html:p>
```

The `dictRef` points to the IUPAC Goldbook [10] temperature [9] and the `units` points to the National Institute of Standards and Technology [10] This illustrates some key virtues of CML/STMML. It can be mixed with text (our ”Datument” approach (REF)) and other markup languages (here HTML) through namespaces and it builds on W3C work (XSD dataTypes). However even though STMML was published 11 years ago there has been very little adoption of *any* markup languages in physical science. There is full support in CML software , including FoX [11].

For this chapter we'll use CML $^{\star\star}_{\star\star}$ and introduce a few self-explanatory terms: `<molecule>` with `<atom>` and `<bond>` and a `<propertyList>` (measured or computed). A `<property>` has a structureType (`<scalar>, <array>or <matrix>`) annotated with `@dataType` (`xsd:string`, `xsd:integer`, `xsd:double` and annotated with a reference to a dictionary (`@dictRef`). This covers the vast majority of compchem - larger dimensions are supported by using CML pointers into (say) HDF or NETCDF.

Here is how a molecule with atoms and coordinates can be completely described

```
<molecule
  xmlns="http://www.xml-cml.org/schema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:units="http://www.xml-cml.org/schema/units"
  xmlns:compchem="http://www.xml-cml.org/dict/compchem"
  >
  <atomArray>
    <atom id="a1" elementType="O" x3="0.0" y3="0.0" z3="0.0"/>
    <atom id="a2" elementType="H" x3="0.96" y3="0.0" z3="0.0"/>
    <atom id="a3" elementType="H" x3="-0.23" y3="0.93" z3="0.0"/>
  </atomArray>
  <propertyList>
    <property dictRef="compchem:dipole">
      <scalar dataType="units:debye" dataType="xsd:double">1.85</scalar>
    </property>
  </propertyList>
</molecule>
```

[8] http://www.ch.ic.ac.uk/rzepa/codata2/, include CODATA ref

[9] `http://goldbook.iupac.org/T06321.html`

[10] `http://physics.nist.gov/Pubs/SP811/sec08.html\#8.5`

[11] Because MLs map well onto Object languages, FORTRAN needs special support and we thank Toby White and Andrew Walker for writing a FORTRAN library for CML and XML

1.2.3 Dictionaries

Dictionaries are fundamental to semantic and therefore reproducible computing. There is a hierarchy of power

- Give every semantic object or concept a unique ID. Where possible we re-use authorities, so
  - a float is defined as xsd:double (defined by W3C),
  - temperature by IUPAC http://goldbook.iupac.org/T06321.html
  - kelvin (units) NIST http://physics.nist.gov/cuu/Units/kelvin.html

  These fit well into rdf/URI and could be written as `nist:kelvin` and `iupac:T06321` using standard prefix notations

- create a dictionary entry with an id and type, and if possible defitition and description

```
<cml:entry id="electricdipole" dataType="cml:vector3">
  <cml:definition>The electric dipole a molecule</cml:definition>
  <cml:description>Dipole moments in molecules are responsible for the
      behavior of a substance in the presence of external electric fields.
      See http://en.wikipedia.org/wiki/Electric_dipole_moment
  </cml:description>
</cml:entry>
```

- add semantic validation of transformation to the entry. This might be done through OWL ontologies or alternatively by adding CML / Declaratron snippets.

The IUCr dictionaries are an excellent example of community-created dictionaries. There is a core dictionary, applicable to most crystallography and many sub-domain dictionaries such as for proteins, diffraction, powder, etc. We recommend the use of multiple dictionaries as this give a community a chance to create well-developed sub-components and then rationalize later. For example we propose one dictionary per computational code (e.g. for NWChem) and then rationalizing parts of these at a higher communal level where possible.

This illustrates the essentials of semantics. molecule, atom, etc are defined in the XML schema. In addition there are thousands of unit tests in JUMBO [12] which act to resolve possible ambiguities in the words in the schema. Some implicit semantics are unavoidable - in chemistry we have defaulted coordinates to Angstrom units (and this information is omitted in the example); other units can be applied explicitly if required. The use of namespaces and semantic dictionary-based annotation is fundamental to CML (and to the MathML in the Declaratron and the Declaratron itself). Here there are the following:

- `http://www.xml-cml.org/schema`. CML with its domain semantics hardcoded. We can rely on a consistent interpretation of the chemistry

- `http://www.w3.org/2001/XMLSchema`. W3C XSD datatypes $\frac{**}{**}$. Complete semantic description of xsd:double, for example giving min/max values, representations, etc. It is possible to use an XSD toolkit as blackbox to manage these with complete confidence. xsd:date could be normalized with (say) JodaTime.

- `http://www.xml-cml.org/schema/units`. $\overline{**}$ Units are fundamental. Although NIST started a UnitsML nearly 20 years ago it was aimed at a database of units $\frac{**}{**}$. There are no agreed computable semantics for units and we use the ones we proposed in STMML. Note the link later to a CML dictionary of units, which is somewhat ad hoc.

[12]"JUMBO" subsumes CMLXOM and XML Dom base on XOM

- `http://www.xml-cml.org/dict/compchem`. There are many hundreds of essential concepts and property definitions required in compchem. Despite 40+ years of computational chemistry programs there is no communal dictionary of terms, let alone a structured ontology. At the Cambridge SPS meeting we started a call for compchem dictionaries and this is being taken forward in Cambridge, CSIRO, PNNL and Kitware and we hope will spread to a wider community of Materials Science Informatics [⋆].
- `atom, elementType, x3...`. CML elements (e.g. `atom`) represent structured objects and attributes (`elementType`, `x3` represent properties with hardcoded semantics. Thus `@elementType` must be found in a standard periodic table (avoiding the problem of, say, `"W1"` authored for water, misread as tungsten). Similarly `x3` is the Cartesian x-coordinate of an atom in Angstrom; this avoids confusion with x2 (chemical formula) and xFract (crystallography). Legacy formats very commonly ambiguate these concepts causing massive wasted work.
- `property`. This is a property of the molecule (atoms, bonds can also have properties). It uses STMML syntax to define a scalar quantity with defined units, defined dataType and defined semantics (through a linked dictionary). The dictionary can, in principle, contain computable semantics for valiadation and transformation.

1.3 Components for Defining Computation

In this chapter we take the view that scientific computation can be broken down into three components:

- **data** to use in computation. In a chemical calculation, this might consist of structures representing atoms, bonds and molecules, their relationships and any parameterisation.
- **formulae** to be applied to the data—for example, the functional form of a molecular forcefield.
- **computational specifications** detailing how the formulae should be applied, and to which bits of data, e.g. which objects are we computing over, are we computing a single value for a given formula, or is it being used as input to an optimiser etc.

We demonstrate techniques for representing data and formulae in a semantically well defined manner, validated with unit tests. We then introduce a tool for specifying computation which preserves semantics, supports data transclusion and sallows a separation of domain knowledge and programming, allowing the combination of well tested "black box" domain objects with declaratively specified computation.

1.3.1 Black-box Libraries

Because chemistry is stable we have been able to create a black-box library (JUMBO [⋆⋆]). JUMBO was developed to act as a reference implementation for CML, with full unit testing. However that also means it is deployable as a reliable component for the Declaratro — we give an example:

```
public double getDistanceTo(CMLAtom atom2);
```

This returns the Euclidian distance between two atoms (or throws an Exception). (The result is as well-defined in chemistry as a squareRoot is in mathematics). The method is tested with a common set of `fixture`s and tolerances (EPS)

```
  public final void testGetDistanceTo() {
   double d = fixture.atom[0].getDistanceTo(fixture.atom[1]);
  Assert.assertEquals("distance", Math.sqrt(3.), d, EPS);
  d = fixture.atom[0].getDistanceTo(fixture.atom[0]);
  Assert.assertEquals("distance", 0.0, d, EPS);
 }
```

The test not only confirms the correct operation but gives guidance to a human developer. There are other Open libraries (e.g. the Chemistry Development Kit, CDK) which can also be reliably used as BlackBoxes.

JUMBO is developed to compare XML documents which are much more suitable than text whose syntactic equivalence suffers from character encoding, whitespace, line endings. A typical JUMBO test:

```
JumboTestUtils.assertEqualsIncludingFloat(
 "MOPAC", referenceXML, textXML, ignoreWhitespace, 1.0E-6)
```

1.3.2 JUMBOConverters and FoX

Because almost all physical science is in non-semantic form there have been many X-to-Y converters written to get the output of one program into another; in chemistry an excellent example is Openbabel. Obviously conversion can only be provided for those concepts which exist in both programs, or can be generated algorithmically or looked up. The converters are rarely completely and generally do not expose any semantics. We strongly recommend conversion to a validatable semantic form and, for chemistry, provide the JUMBOConverter$\frac{\star\star}{\star}$ framework with currently about 60 programs being converted to CML.

Converters are often written where the source code is not visible so we have to guess. There are two methods:

- Procedural, using Python, Java, C++ etc. The problem is that the semantics are not visible and the tools do not validate against dictionaries.

- JUMBOConverter templates $\frac{\star\star}{\star}$. Here the legacy output is mapped semantically onto the validatable result. This forces the translator to define the dictionary entries used and also leads to a robust implementation of unit and regression tests. It is also suited to analysing large corpora.

1.3.3 MathML

MathML$\frac{\star\star}{\star\star}$ is a W3C math working group recommendation for representing mathematics on the Web. At the time of writing, this is an emerging web standard, with some level of support in popular browsers, and an ecosystem of supporting tools such as MathJax[13] to aid inline display. Although browser support is transitional, on the authoring side there are many tools available: the W3C lists more than 30 editors and viewers[14].

There are two distinct dialects of MathML, with different goals:

[13] `http://www.mathjax.org/`

[14] `http://www.w3.org/Math/Software/mathml_software_cat_editors.html`

Display MathML represents the visual layout of mathematical equations. It concerns itself with how symbols are displayed and arranged on the page, but not what they *mean*.

Content MathML [3] is oriented towards represeting the semantics of mathematics: "to provide an explicit encoding of the underlying mathematical meaning of an expression" [7].

Content MathML$^{\star\star}_{\star}$ (CoMML) is highly suited to representing computation in a semantically aware manner - it has been designed to add an extra layer of formalism to the communication of mathematical equations, removing several sources of potential ambiguity.

1.3.4 Executable MathML

In order to use CoMML in an executable document it must be linked to a tool which can run the computations which it encodes. Here we use SCMathML[15], a Scala engine for running computations specified using CoMML. Scala is a functional language which runs on the Java VM, combining the power of functional programming with easy interoperability with Java code. It was used here because:

- it is very concise - most of the MathML entities are defined in about a hundred lines of code;
- inbuilt XML support makes starting to work with XML easy - in the listings below, XML blocks define real Scala objects, rather than strings to be parsed;
- support for building Domain Specific Languages, with flexible parsing.

SCMathML 'parses' CoMML into a tree of Scala objects which can carry out computation. We will illustrate, using unit tests from the framework, how this works, and how it can be used for computation in the physical sciences. Unit tests are written using the ScalaTest framework, which—along with some wrapper functions—leads to clean, self-documenting test code: all of the code examples below are taken directly[16] from the Unit Test files. For example, to check that a MathML `<cn>` element is parsed into a SCMathML constant we can write:

```
parsing(<cn>5.3</cn>) should equal( DoubleConstant(5.3))
```

Two of the basic elements of MathML are constant numbers (`<cn>` for content numeric) and variables (`<ci>` for content identifier). Variables need to have values provided if a function is to be computed - for example, when given $y = x^2 + c$, if we want to get a number out, we need to provide values for both x and c. In SCMathML, this is done through a *Context*, where objects can be passed in:

```
evaluating(<ci>x</ci>, "x"->5) should equal( 5 )
```

In this example "evaluating" is a function defined to take a MathML expression, and some mappings of strings to objects, and evalute the expression. `should` and `equal` (and later `be`, `plusOrMinus`) are ScalaTest functions which allow a natural reading of unit tests. This example can be read that: if we take the expression `<ci>x</ci>`, parse it, and then evaluate it in a context where x has been set to 5, we should get 5 out. This is an illustration of how objects are *bound* to variables, and used to evaluate abstract mathematical expressions and obtain concrete results.

CoMML is strongly influenced by Scheme and related languages: it uses `<apply>` tags to denote function application, with the first argument being the function to apply. For example

[15] `www.mo-seph.com/projects/SCMathML`

[16] Formatting has been changed, and some variables have been renamed for clarity out of context

`<apply><plus/><cn>2</cn><cn>2</cn></apply>` is roughly equivalent to `( plus 2 2 )` in Scheme. Or:

```
evaluating( <apply><sin/><ci>x</ci></apply>, "x"->3 )
   should equal( Math.sin(3))
```

For a slightly larger example, we can implement Leibniz's method of approximating π,

$$\sum_{k=0}^{n} \frac{(-1)^k}{2k+1} \approx \frac{\pi}{4}$$

```
evaluating(
<apply><times/><cn>4</cn>
  <apply><sum/>                     <!--carry out a summation -->
    <bvar><ci>k</ci></bvar>         <!-- for k in ... -->
    <lowlimit><cn>0</cn></lowlimit> <!-- start at 0 -->
    <uplimit><ci>n</ci></uplimit>   <!-- go up to the value bound to n -->
    <apply><divide/>
      <apply><power/><cn>-1</cn><ci>k</ci></apply>
      <apply><plus/>
        <apply><times/><cn>2</cn><ci>k</ci></apply>
        <cn>1</cn>
      </apply>
    </apply>
</apply></apply>, "n"->4000) should be ( 3.1415 plusOrMinus 0.01 )
```

Finally, there is often a need to work with values obtained from domain entities. In order to do this, we have defined a small set of extensions to the MathML specification to interface with existing objects, using the `<csymbol>` tag. Figure 1.2 gives a worked example where values are extracted from domain objects. It is based on Hookes law, which would typically be written as:

$$E = \sum_{bonds} \frac{1}{2} k x^2$$

However, since it has been translated into MathML, with bindings added, it is clear that:

1. x actually refers to displacement from equilibrium length, and so has to be split into l and l_0.
2. all of the values are specific to a given spring, including the spring coefficient k

Contrast this with the first term in the AMBER forcefield equation (Figure 1.1) where it is up to the reader to interpret that that i) b is a subscript for the current bond ii) b on its own means the length of the current bond iii) b_0 is the equilibrium length *of the current bond*.

It should be noted, however, that this example only defines the *mathematical* semantics—the operations to be carried out, and the bits of data to appply them to. It does not deal with any *domain* semantics, as the Springs class is not semantically explicit.

```
var sum = //Define Hookes law in Content MathML
<apply>
  <sum/>
  <bvar><ci>spring</ci></bvar>     <!-- this is the variable to bind -->
  <condition>                      <!-- and this is the set to bind over -->
    <apply><in/><ci>spring</ci><ci type='set'>springs</ci></apply>
  </condition>
  <apply><times/>
    <cn>0.5</cn>
    <apply><times/>
      <apply><csymbol function='elasticity'>k</csymbol><ci>spring</ci></apply>
      <apply><power/>
        <apply><minus/>
          <apply><csymbol function='length'>l</csymbol><ci>bond</ci></apply>
          <apply><csymbol function='equilibrium'>l0</csymbol><ci>bond</ci></apply>
        </apply>
      <cn>2</cn>
    </apply>
  </apply>
</apply>
</apply>
//We have have a spring class which takes 3 arguments:
//length, equilibrium length, and elasticity
//Create two Springs to test:
var bonds = List(new Spring("A",8,7,3),new Spring("B",10,9,4))

// Target equation is: sum of
// 0.5 * elasticity(spring) * (length(spring)-equilibrium(spring))^2
// With the test springs, the expected value is:
val exp =
    0.5 * 3 * Math.pow(8-7, 2) + //Spring A
    0.5 * 4 * Math.pow(10-9, 2); //Spring B

//Now run the test:
evaluating( <math>{sum}</math>, "bonds"->bonds ) should equal( exp )
```

FIGURE 1.2
Example MathML equation, showing a test summation over a set of domain specific objects. Target equation is $E = \frac{k(l-l_0)^2}{2}$. Note that this example is not semantically bound.

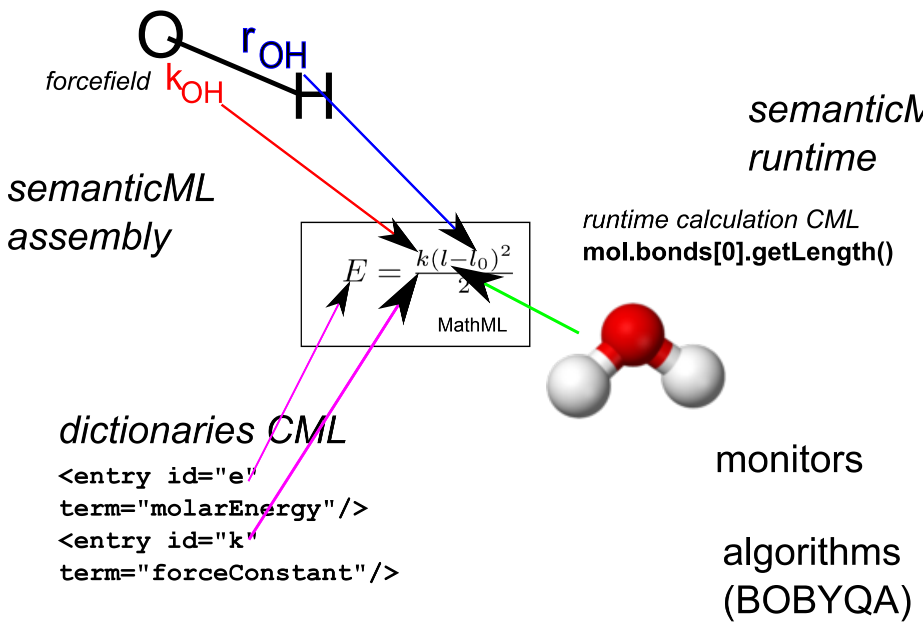

FIGURE 1.3
Declarative creation and execution of `computationalDocument`. The mathematics is linked to domain sematics (e.g. for each element of the summation `l` links (arrows) to reference equilibrium bond-length `l`$_0$*inforcefield(e.g.foranO − Hbond)and*`k`*tothereferenceO − Hforce − constant.Dictionariesareusedtoensuresemanticmapping(e.g.thatEnergycanberelatedtoforce− constantsandlengths)andalsotoprovidehumanprose.ThedocumentisassembledandmodifiedbyrepeatedbyXPathquer friendly(minimal)documentisautomaticallyexpandedtoconstancyandmachine − interpretability.Inthesecondphasethedocumentisexecutedusingvistorstotraversethe(nowconstant)treeandprocessn pointoptimizerBOBYQA).*

1.4 Semantic Physical Computation

We have discussed the representation of scientific entities using domain specific markup languages, and formalisation of computation using domain independent markup. Now, we introduce a system—"The Declaratron" **—which brings these together to carry out reproducible scientific calculations.

The Declaratron consists of:

1. An XML dialect for specifying declarative computation. Our current vocabulary is:

   - `<sem:computationalDocument>` the overall container and organizer;
   - `<sem:editor>` which allows the document to modify itself using copy, transform, move and delete operations;
   - `<sem:assert>` allows components to be tested against scalar values or complete (XML) files;
   - `@href` allows input of files (transclusion-copy);

- •`<sem:writer>` allows output of sections of the document;
- •{`<sem:functionalForm>` specification of a MathML expression which can be bound to other domain semantics;
- •`<sem:computation>` evaluation of a `<sem:functionalForm>` either once or in an algorithm.

2. Core libraries that support the operations in this dialect: transclusion, copying, merging, validation.
3. support for replacing XML nodes with domain objects that bring useful code with them (decoration). This includes decorating data structures (e.g. replacing a plain `<cml:atom/>` element with a Java object, and creating executable objects, such as unit converters.
4. links to domain libraries to bring in necessary semantics and computational elements, including:
   - •SCMathML for evaluating mathematical formulae in the context of a scientific computation
   - •general STM information, such as units and their conversions
   - •CML/JUMBO for representing chemical data and computing common properties

   It is entirely possible for users to add their own domain libraries.

Much of the power of the declaratron comes through the data structure of XML and XPath - the latter being a formal language for addressing components of an XML document. Xpath can reference any set of nodes in the tree (nodeSet) with a natutral and powerful syntax (based on tree structures). A series of editor commands means documents can be modified (including self-modification). XML acts as both input and output, and can provide a full record of computations. Snippets from files such as schemas and dictionaries can be included in the output so that it is clear exactly what versions of what were used and what was done.

The Declaratron works as follows, illustrated in Figure 1.4:

1. read in a computational document;
2. manipulation: transclusion and substitution (see Section 1.4.1);
3. decoration: replacing standard XML elements with domain objects;
4. computation.

Validationis threaded through the entire process, to check that incoming data is in the correct form, manipulated data has the right properties, and the results of computations are correct.

We will use a case study—calculating the energy of water using a very simple forcefield—to demonstrate this.

1.4.1 Bridging the gap between human- and machine-readable semantics

Humans work with implicit semantics, and require documents to be small and non-repetitive in order to extract meaning. Machines require explicit, formal semantics, and can work with large (typically 10-100 times larger), repetitive documents that are deeply human-unfriendly.

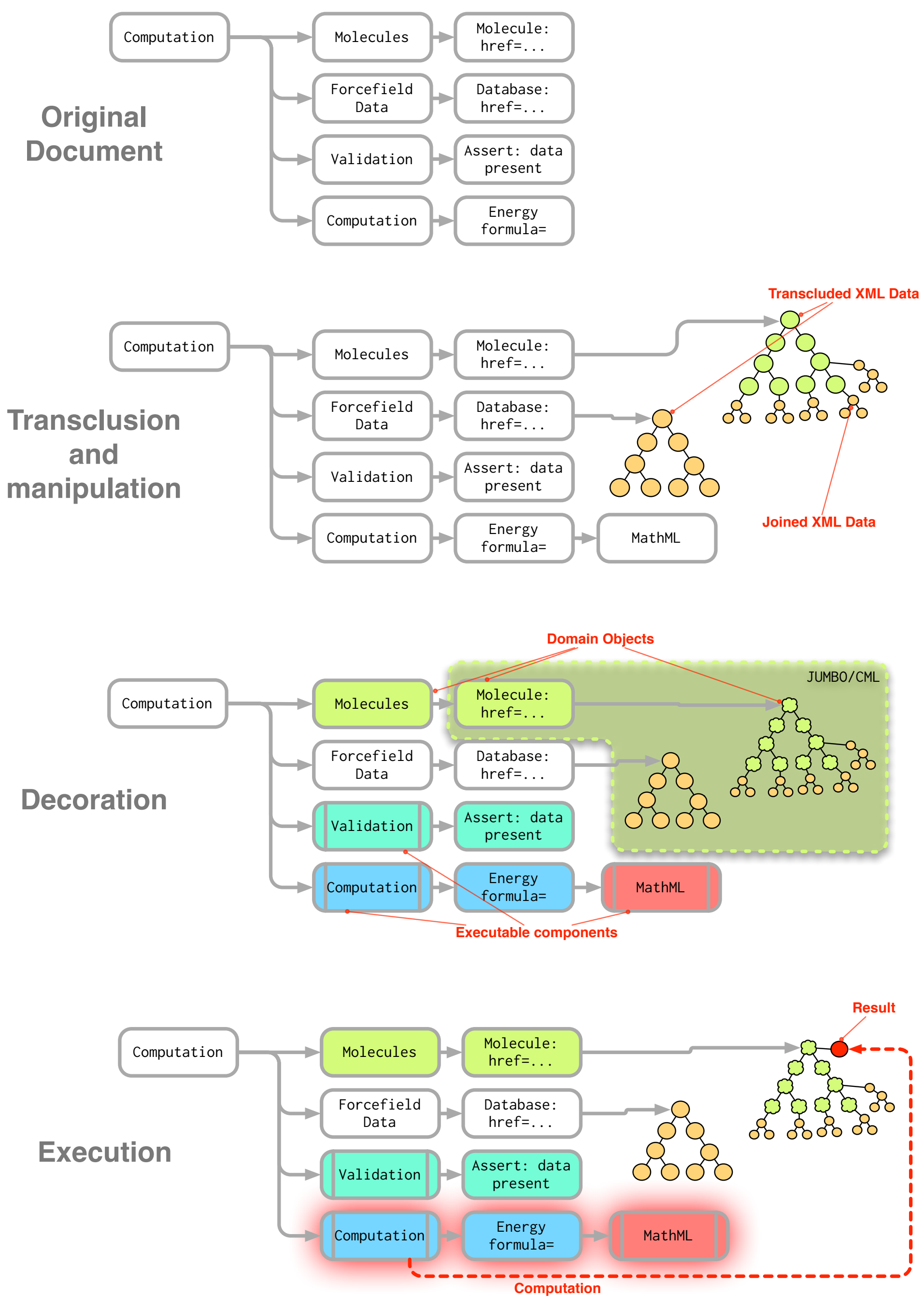

FIGURE 1.4
Overview of Declaratron operation. i) original document; ii) manipulated XML document iii) decorated document with executable domain objects iv) executing a computation

In order to address this division, the Declaratron can use documents which are written in a relatively concise, human readable form, and automatically expanded to the complete, explicit computable form. The human form uses:

- Key-value syntactic substitution, which both reduces repetition, and allows for human readable names to be attached to complex structures. For example, replacing occurences of `xpath="//cml:molecule[@id='molecule']"` with `xpath="${molpath}"` makes the document more readable as the intent of the XPath expression is clear, and makes it more robust as complex expressions can be defined and tested once, and then re-used.
- Transclusion of files; there is a `<sem:editor>` which expands `href` attributes recursively. Again, this enhances readability, especially where large files are brought in, and re-used, as common elements can be put into files and shared (e.g. `<maths href="${mathsPath}/hookesLaw.xml"/>`. It also allows the use of data from non-local sources, as any URI which provides an XML stream can be used.

Using this system of transclusion we have successfully computed the energy of acetic acid (8 atoms) using the current AMBER (PARM94) forcefield, by expanding a human readable input, to machine form and evaluating it. However the human form hides many important details, and for this chapter we have therefore chosen a very simple complete example (water — 3 atoms), using a highly simplified functional form (only bonds), with a later indication of how this would be expanded to a more complete example.

1.4.2 Example: Water — Energy Calculation

To illustrate the Declaratron's operation, we will walk though a calculation file step by step$\frac{\star}{}$. This file carries out single point energy computation and atomic optimisation on water, using the Hooke's law term from the AMBER forcefield ($\sum_{bonds} K_b(b-b_0)^2$) and associated parameters[17]. This is described in four stages:

- setting up the document: namespaces and named variables;
- identifying the data to be used: the molecule, atoms and bonds;
- specifying the forcefield;
- specifying the computations to be carried out.

In a real-world file, the definitions of data and computations might be given in a different order—for example, putting the computation first in files makes it easy for a human to find out what a file does. The order of elements is fairly flexible, so in this explanation we start with the data and then later specify what we intend to compute. As noted previously, we also go through some elements which would typically be expanded from databases, or refactored into individual files to make a humane document.

1.4.2.1 Set up the document

The `<computationalDocument>` node is a container for the entire computation. We also define XML namespaces which can be used throughout the file:

```
<computation xmlns="http://www.xml-cml.org/semanticcomputation"
   xmlns:m="http://www.w3.org/1998/Math/MathML"
   xmlns:cml="http://www.xml-cml.org/schema"
   xmlns:amber='http://www.xml-cml.org/dict/amber:gaffType'>
```

[17]The full file can be found at: https://bitbucket.org/petermr/semantic-forcefield/src/cf7ef9b03020/src/main/resources/org/xmlcml/cml/examples/amberNew.xml?at=default

In order to carry out XPath queries over namespaced documents, it it necessary to setup namespace prefixes which can be used by the XPath engine:

```
<!-- setup XML namespaces for use in XPath queries -->
<queryNS prefix="semc" uri="http://www.xml-cml.org/semanticcomputation"/>
<queryNS prefix="semf" uri="http://www.xml-cml.org/semanticforcefields"/>
<queryNS prefix="cml" uri="http://www.xml-cml.org/schema"/>
<queryNS prefix="m" uri="http://www.w3.org/1998/Math/MathML"/>
<queryNS prefix="amber" uri="http://www.xml-cml.org/dict/amber:gaffType" />
<queryNS prefix="cmlx" uri="http://www.xml-cml.org/schema/cmlx" />
```

Throughout the file, key/value pairs can be used to reduce repetition. Here we set up variables for XPath queries for i) the molecule to analyse, using an `id` to ensure only the desired molecule is used; ii) finding all the bonds belonging to that molecule; iii) finding all the atoms:

```
<!-- XPath references to molecule, atoms, bonds -->
<keyValue name="molpath" value="//cml:molecule[@id='molecule']"/>
<keyValue name="bondpath" value="${molpath}/cml:bondArray/cml:bond"/>
<keyValue name="atompath" value="${molpath}/cml:atomArray/cml:atom"/>
```

1.4.2.2 Identify and verify chemical data

Now we define the chemical data to be used. Firstly, the molecule. As this is a container for chemistry, it defines some extra namespaces for chemical entities:

```
<cml:molecule
  xmlns:cml="http://www.xml-cml.org/schema"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:units="http://www.xml-cml.org/schema/units"
xmlns:compchem="http://www.xml-cml.org/dict/compchem"
>
```

Next, the atoms. Note that each atom has a *atomType* defined by AMBER. This is *not* the same as the elementType—AMBER uses atomTypes for as well as elementTypes for atoms, the distinction being that atomTypes change depending on the surrounding atoms. For example, an oxygen bonded to a hydrogen has `atomType "OH"`. Bond parameters are looked up using atomTypes rather than elementTypes, and bonds are often annotated with different atomTypes in different communities of practice (i.e. different laboratories).

```
<cml:atomArray>
  <cml:atom id="a1" elementType="O" x3="0.0" y3="0.0" z3="0.0">
     <atomType dictRef="amber:parm94Type">OH</atomType>
    </cml:atom>
  <cml:atom id="a2" elementType="H" x3="0.96" y3="0.0" z3="0.0">
      <atomType dictRef="amber:parm94Type">H</atomType>
    </cml:atom>
  <cml:atom id="a3" elementType="H" x3="-0.23" y3="0.93" z3="0.0">
      <atomType dictRef="amber:parm94Type">H</atomType>
    </cml:atom>
</cml:atomArray>
```

The `id`-structure is very important and is used to link components of the document (e.g. the bonds reference the atom-`id`s) or even to aggregate them (through the editor).

The final component of the chemical data is a set of bonds, with parameters looked up by atomType. For each bond, `k` is the spring constant, `req` is the equilibrium bond length and `desc` is the formal description (e.g. a literature reference).

```
<cml:bondArray>
  <cml:bond atomRefs2="a1 a2">
      <cml:property>
          <cml:list id="c_ct" cmlx:atomTypesId="OH__H">
```

```
                <cml:atomType dictRef="amber:parm94Type">OH</atomType>
                <cml:atomType dictRef="amber:parm94Type">H</atomType>
                <cml:scalar dictRef="ff:k" dataType="xsd:double">317.0</scalar>
                <cml:scalar dictRef="ff:req" dataType="xsd:double">1.522</scalar>
                <cml:scalar dictRef="ff:desc" dataType="xsd:string">JCC,7,(1986),230;AA</scalar>
            </cml:list>
        </property>
    </cml:bond>
    <cml:bond atomRefs2="a1 a3"> <!-- contents omitted for brevity --> </cml:bond>
  </cml:bondArray>
```

For the purposes of this example, we have declared the bond properties inline. In general, these would be pulled in from a knowledgebase automatically, but this is too complex for this chapter.

Once all of the data is in place, we can verify it. To ensure that all atoms can be annotated with the AMBER parm94 dictionary, an `<assert>` element checks that i) there are 0 atoms without a valid `id` and ii) there are 0 atoms without a valid `atomType`:

```
  <!-- all atoms in the document must have ids and atomTypes (expressed as negation) -->
  <assert count="0" xpath="${atompath}[not(@id)]"/>
  <assert count="0" xpath="${atompath}[not(cml:atomType[@dictRef='amber:parm94Type'])]"/>
```

1.4.2.3 Specify the forcefield to be used

After specifying the data, we specify the functional form of the forcefield ($E = \frac{k(l-l_0)^2}{2}$). It starts wuth the summation over bonds

```
 <functionalForm id="hookes"
  hrefSource="src/main/resources/org/xmlcml/cml/forcefield/functional/harmonicBond.xml"
  xmlns="http://www.xml-cml.org/semanticforcefields">
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <apply><sum/>                   <!-- Sum -->
      <bvar><ci>bond</ci></bvar>      <!-- For bond -->
      <condition>                   <!-- in bonds -->
        <apply><in/><ci>bond</ci><ci type="set">bonds</ci></apply>
      </condition>
      <!-- This is what is inside the sum -->
      <apply><times/><cn>0.5</cn>   <!-- divide by 2 -->
```

We need to bind the value of k to the actual bond (`property` is child of `bond`). Again, note the use of `dictRef`—this uses a defined id reference, whch means that the semantics of the value to be used can be looked up in the dictionary:

```
        <apply><times/>                <!-- get k for the current bond -->
          <apply>
            <csymbol xpath="./cml:property/cml:list/cml:scalar[@dictRef='ff:k']">k</csymbol>
            <ci>bond</ci>
          </apply>
        </apply>
        <apply><power/>                <!-- start the squared term -->
          <apply><minus/>              <!-- start l-l_0 -->
```

The value of l is bound to the result of calling the JUMBO function `cml:bond.getBondLength()` for each bond:

```
            <apply>
              <csymbol function="getBondLength">l</csymbol>
              <ci>bond</ci>
            </apply>
```

and the reference (equilibrium) length is looked up for each bond with a XPath expression that selects the relevant property from its descendants.

```
          <apply>
            <csymbol xpath="./cml:property/cml:list/cml:scalar[@dictRef='ff:req']">l0</csymbol>
            <ci>bond</ci>
          </apply>
        </apply>                    <!-- end l-l_0 -->
        <cn>2</cn>                  <!-- end of the squared term -->
      </apply>
    </apply>
  </apply>
</math>
</functionalForm>
</computation>
```

At this point, we have defined a molecule, and the forcefield which is to be applied to it.

1.4.2.4 Specify the computation to be carried out

We now specify a computation co carry out with these entities (do we want simply to evaluate the energy, or adjust the geometry to optimze the structure against its energy?). First, let's create a node (child of `molecule` to hold the result of a single point energy calculation:

```
  <editor method="createChild" xpath=".//molecule" element="cml:scalar" targetId="singlePoint"/>
```

Next, specify that an evaluation of the functional form with the molecule in its initial configuration should be carried out. This will locate the functional form and ask it to evaluate itself, using the molecule as input. When the molecule is passed in, the set of bonds will be bound to the variable `bonds`. The functional given above will iterate over the set of bonds, and for each bond call the JUMBO function to find the current bond length, subtract the equilibrium length etc. as detailed above. After the calculation, we ensure that the output has the correct value, and has the correct units:

```
  <computation method="singleEvaluation"
    formula="//functionalForm[@id='hookes']" input="${molpath}">
    <variable name="bonds" xpath="${bondPath}"/>
  </computation>
  <assert value="1.234" xpath=".//scalar[@id='singlePoint']"/>
  <assert value="units:joule" xpath=".//scalar[@id='singlePoint']@units"/>
```

The optimum geometry of a molecule is that of lowest energy and many calculations attempt to find this using a variety of algorithms. We have chosen the recent BOBYQA method [17] which does not require analytical derivatives (or second derivatives).

The same functional form can also be used in the optimisation of geometry to find the minimum energy. Here we use the non-derivative optimiser BOBYQA by giving it i) a target function to evaluate (the functional form); ii) the data to work over (the molecule); and iii) an XPath expression to find the free variables in the optimisation. The optimisation happens *in place*, so the modified atom positions are now part of the document:

```
  <computation method="optimise" algorithm="BOBYQA"
    formula="//functionalForm[@id='hookes']" input="${molpath}"
    freeVariables=".//@x3 or .//@y3 or .//@z3"/>
```

Finally, we can compare the output geometry and energy with the a previous computation stored in another file. This is a complete nodewise comparison of XML, which will ensure both semantic identity and numerical identity, including a tolerance (`eps`) for floating point variations.

```
  <assert href="expected.xml" ref="${molpath}" eps="1.0E-06"/>
</computation>
```

1.4.3 Moving beyond toy examples

The Declaratron has a wider range of features than we have illustrated here. There is a one-off cost to transforming legacy files to CML (some of which can be done with JUMBOConverter-templates). Most problems then require a complex process of locating transcludable information (see Section 1.4.1), extracting the desired nodes and inserting into the growing `semanticDocument.` Although this is a complex operation, once constructed, the semanticDocument subtrees can be re-used without change for future computations. This means that computations can be stated very simply in terms of the major free variables and the operations to be performed on them.

As an example, to use the parm94 database in a semantic calculation, we would carry out the following steps:

First, the JUMBO atomTypeTool can be used to add the required ids:

```
<!-- list is the default type for general data -->
<cml:list id="parm94Test" href="${forcefield}/amber/parm94test.xml"/>
<!-- transform (add id) to atomType children -->
<atomTypeTool method="addAtomTypesId" using="./cml:atomType"
    xpath="//cml:list[@id='parm94Test']/cml:list/cml:list[count(cml:atomType)>0]" />
```

Then we can merge functional form and parameters for each bond from the database, by copying the relevant information into each `cml:bond` element:

```

<moleculeTool method="getOrCreateBonds" xpath="${molpath}" setId="createdBonds"/>
<editor method="copyChild"
    xpath="//cml:molecule[@id='molecule']/cml:bondArray"
        from="//cml:list[@id='parm94Test']/cml:list[@title='bonds']/semf:functionalForm" />
```

We can also ensure that the bond angles are in the correct units:

```
<unitsVisitor xpath="//cml:list[@id='parm94Test']//cml:scalar[@dictRef='ff:angeq'
  or @dictRef='ff:phase']" method="degrees2Radians"/>
```

And finally, we can save this annotated molecule into its own file—coarse-grain memoization—so we do not have to do the conversion again in the future:

```
<writer xpath="//cml:list[@id='parm94Test']" file="output/parm94testNew.xml" />
```

All this can be packaged into standard operations—a file can be created for any given conversion and transcluded where necessary—so that the final calculation mirrors the human readable form, and is expanded into the detailed semantic form at runtime.

The program output can contain as detailed a list as we like of the operations and their outputs/results. It could contain fine grained information for debugging or simple summary data. It will have a complete record of the input - not just the values but the semantic parameters, the dictionaries and the functional forms. This means the output is immediately re-runnable. Note that XML has a very wide range of Open document maniuplation tools so we can build high-quality print or semantic indexes.

The output is directly transformable into the inputs of other programs which share some or all of the semantics (e.g. chemistry and mathematics). In many cases these can be understood without the wider context — a molecule optimised by a forcefield could then be read into a QM program or posted in an online CML repository for use in chemical informatics.

1.4.4 Integrating Semantic Physical Computation with emerging architectures and automation

Figure 1.5 shows an example computational chemistry (CompChem) architecture. The input (LHS) consists of about 6 orthogonal axes: i) Molecules; (ii) Commands —the scientific

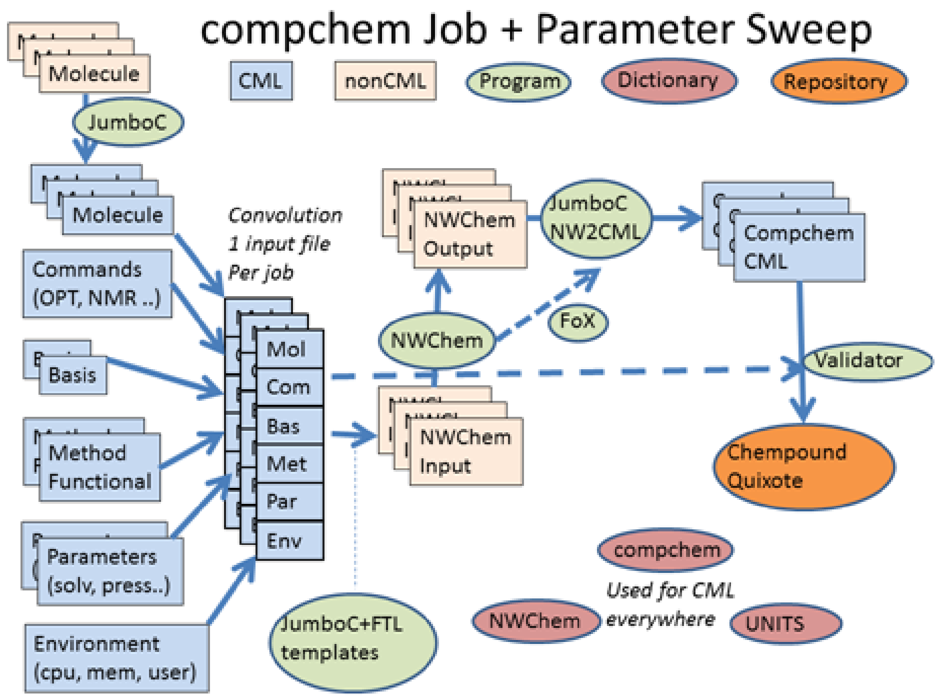

FIGURE 1.5
A multi-axis parameter sweep for computational chemistry using NWChem** as a semantic framework * for computing properties of matter. The experiments often involve several axes and could involve many thousands of jobs. Our architecture allows the automatic creation of jobs with combinations of settings ("parameter sweeps") which can be filtered semantically. NWChem outputs CML through FoX calls; alternatively the legacy output is converted by an NWChem JUMBO-comverter to CML. The output is semantically validated against (a) NWChem (b) General Compchem (c) Units dictionaries and normalized, before archive and redistribution in a Quixote repository [5]. Note: We thank PNNL for making NWChem Open and agreesively CML-ising it.

problem to be solved; iii) Basis-Set—the quantum mechanical parameterisation; iv) Method of solving QM equations; v) Physical Parameters, e.g. temperature, pressure; vi) Computer Environment, e.g. CPU limits, memory, number of processors. While this is a particular example, and would be used for experiments such as "run 1000 molecules with 3 basis sets", many experiments have a similar structure to this, i.e. "explore a defined subset of the parameter space".

The example in Figure 1.5 has several independent input axes and it is clear that these must be semantically defined if the experiment is to be reproducible—ambiguity or the possibility for disagreement about the meanings or intents of specifications will result in different or unpredictable results. More generally, no science can be reproducible without agreed semantics.

The Declaratron is very well suited to the flexible generation of input — it allows parameter axes to be declared semantically and combined through domain-specific commands (expanding the scope and precision of "Convolution"). It is also a critical part of marshalling and validating output, especially transforming documents to have different structures and components.

1.5 Conclusions

The adoption of semantics by long-tail physical science has been extremely slow and its absence causes millions of lost hours and costs hundreds of millions of dollars. We do not believe science is reproducible without a committed community (as in crystallography or astronomy or much of bioscience).

Instrumental and sensor output is now massive, but there are very few semantic implementations or dictionaries; this is an essential task for the communities to tackle.

In this chapter we have demonstrated a system which addresses several of these concerns. It integrates existing semantically aware components with legacy data, and provides a strongly semantically grounded computation environment, with a high level of reproducibility.

The usefulness of any semantically aware, reproducible system is dependent on its context: the further the semantic frontier is pushed back, the the more use we can make of each component in the system. To make this happen we have a set of recommendations:

1. Build a community of practice around semantic computation and reproducibility. Ideally this should be through learned societies or International Scientific Unions — without community semantics there is no interoperability and hence no effective reproducibility.
2. Adopt STMML semantics where possible: using strongly typed quantities with dictionaries. Dictionaries can be created in a semi-formal manner, being ratified by formal groups when they are proved to work.
3. Create black-box libraries for fundamental domain-specific operations and algorithms. These should be tested using declarative approaches, to allow for integration with semantic systems.
4. Build declarative validators for legacy code bases (e.g. AMBER) so that their correctness can be verified on a wide range of archetypal problems.
5. Make all "documentation" semantic and computable: write examples in manuals as executable code, so that the documentation is always in sync with the code.

Bibliography

[1] I David Brown and Brian McMahon. CIF: the computer language of crystallography. *Acta Crystallographica Section B: Structural Science*, 58(3):317–324, 2002.

[2] Geoff Brumfiel et al. Beautiful theory collides with smashing particle data. *Nature*, 471(7336):13–14, 2011.

[3] David Carlisle. OpenMath, MathML, and XSL. *ACM SIGSAM Bulletin*, 34(2):6–11, 2000.

[4] Antony N Davies and Peter Lampen. JCAMP-DX for NMR. *Applied spectroscopy*, 47(8):1093–1099, 1993.

[5] Pablo de Castro, Pablo Echenique, Jorge Estrada, Marcus D Hanwell, Peter Murray-Rust, and Jens Thomas. The quixote project: Collaborative and open quantum chemistry data management in the internet age.

[6] MT Dove, AM Walker, TOH White, RP Bruin, KF Austen, I Frame, GT Chiang, P Murray-Rust, RP Tyer, PA Couch, et al. Usable grid infrastructures: practical experiences from the eMinerals project. In *Proc. UK e-Science All Hands Meeting 2007*, pages 48–55, 2007.

[7] W3C Math Working Group. Content MathML. `http://www.w3.org/TR/MathML3/chapter4.html`.

[8] Peter Lampen, Heinrich Hillig, Antony N Davies, and Michael Linscheid. JCAMP-DX for mass spectrometry. *Applied spectroscopy*, 48(12):1545–1552, 1994.

[9] Robert S McDonald and Paul A Wilks. JCAMP-DX: A standard form for exchange of infrared spectra in computer readable form. *Applied Spectroscopy*, 42(1):151–162, 1988.

[10] Alan D McNaught and Andrew Wilkinson. *Compendium of chemical terminology*, volume 1669. Blackwell Science Oxford, 1997.

[11] Peter Murray-Rust. Semantic science and its communication-a personal view. *Journal of Cheminformatics*, 3(1):1–7, 2011.

[12] Peter Murray-Rust and Henry S Rzepa. Chemical markup, XML, and the Worldwide Web. 1. Basic principles. *Journal of Chemical Information and Computer Sciences*, 39(6):928–942, 1999.

[13] Peter Murray-Rust and Henry S Rzepa. STMML. A markup language for scientific, technical and medical publishing. *Data Science Journal*, 1:128–192, 2002.

[14] Peter Murray-Rust and Henry S Rzepa. Semantic Physical Science. *Journal of Cheminformatics*, 4(1):1–7, 2012.

[15] KS Noll, MA McGrath, LM Trafton, SK Atreya, JJ Caldwell, HA Weaver, RV Yelle, C Barnet, and S Edgington. HST spectroscopic observations of Jupiter after the collision of comet Shoemaker-Levy 9. *Science*, 267(5202):1307–1313, 1995.

[16] Shyue Ping Ong, Anubhav Jain, Geoffroy Hautier, Michael Kocher, Shreyas Cholia, Dan Gunter, David Bailey, David Skinner, Kristin A Persson, and Gerbrand Ceder. The Materials Project, 2011.

[17] Michael JD Powell. The BOBYQA algorithm for bound constrained optimization without derivatives. *Cambridge NA Report NA2009/06, University of Cambridge, Cambridge*, 2009.

[18] Alexander Roth, Ronny Jopp, Reinhold Schäfer, and Gary W Kramer. Automated generation of AnIML documents by analytical instruments. *Journal of the Association for Laboratory Automation*, 11(4):247–253, 2006.

[19] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Hubertus JJ Van Dam, Dunyou Wang, Jaroslaw Nieplocha, Edoardo Apra, Theresa L Windus, et al. NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.