JOURNAL OF CHEMICAL INFORMATION AND MODELING

Article

pubs.acs.org/jcim

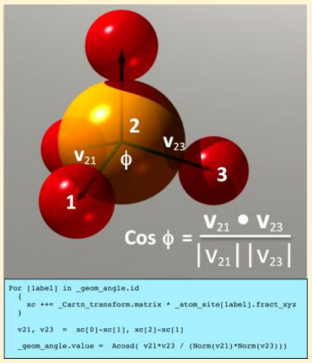
dREL: A Relational Expression Language for Dictionary Methods

Nick Spadaccini,* Ian R. Castleden, Doug du Boulay, and Sydney R. Hall*

School of Chemistry and Biochemistry, The University of Western Australia, Nedlands 6009, Australia

Supporting Information

ABSTRACT: The provision of precise metadata is an important but a largely underrated challenge for modern science [*Nature* **2009**, *461*, 145]. We describe here a dictionary methods language dREL that has been designed to enable complex data relationships to be expressed as formulaic scripts in data dictionaries written in DDLm [Spadaccini and Hall *J. Chem. Inf. Model.* **2012** doi:10.1021/ci300075z]. dREL describes data relationships in a simple but powerful canonical form that is easy to read and understand and can be executed computationally to evaluate or validate data. The execution of dREL expressions is not a substitute for traditional scientific computation; it is to provide precise data dependency information to domain-specific definitions and a means for cross-validating data. Some scientific fields apply conventional programming languages to methods scripts but these tend to inhibit both dictionary development and accessibility. dREL removes the programming barrier and encourages the production of the metadata needed for seamless data archiving and exchange in science.



INTRODUCTION

Information science depends implicitly on the availability of metadata that precisely types, classifies, and relates data items in a form suitable for automatic evaluation and validation. Most formalized approaches to metadata in use today, such as the XML Schema,[3] define the syntactic and structural characteristics of data rather than the relationships that are important to their meaning and definition, and ultimately to their evaluation and validation. When more formulaic relationships have been included in data dictionaries, they are usually encoded in conventional computer languages that are relatively obscure for nonprogrammers. An additional difficulty for conventional computer languages when defining relational information is that considerable coding is needed for the "bookkeeping" aspects of algorithms. This further limits easy access to their definition. Together these represent a serious obstacle to the routine availability of, and access to, data interdependencies and relationships in data dictionaries.

We believe that the inclusion of data relationships in data dictionaries needs to satisfy the following basic requirements. First, the relationships must be presented in a simple *canonical form* that can be readily written and understood by nonprogrammers. Second, the canonical expressions should be amenable to seamless reverse translation into textual formulas; and third, if desired, the relationships should be executable computationally for any particular instance of data. The dREL language described in this paper meets these three requirements.

The basic design for the prototype dREL language was first proposed[4] a decade ago. It is a definitional language designed specifically for data stored in the STAR data model[5] and defined in a dictionary written in DDLm.[2] The STAR data model represents data as simple tag−value pairs and as loops of category data that are equivalent to relational tables.

The ability to define complex data relationships as simple text formulas will greatly advance the development of data dictionaries in many domains.[6] Most scientific fields are interested and capable of undertaking the effort needed to define domain data provided the dictionary language poses no major barriers to metadata construction. Unfortunately, at the moment, the data definition field is akin to the early days of automobiles when most drivers were also expected to be skilled car mechanics. Cars today are endemic because this is no longer the case; while today's car is much more complex "under the hood/bonnet", the driving environment is tailored to the nonmechanic. Equivalent design criteria are needed for data dictionaries, and we assert that a data definition language that allows complex data relationships to be expressed in a simple canonical form will go a long way toward accommodating and attracting the data "driver" as well as the data "mechanic".

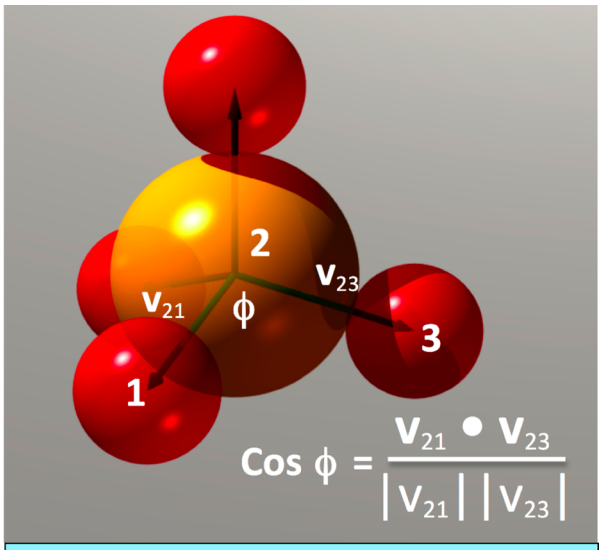
In this paper we describe the methods language dREL for defining data relationships in a canonical form that can be written and understood with little or no programming experience. The language is sufficiently rich and complete that it is amenable to automatic execution. A prototype engine for executing dREL has been tested against a set of crystallographic dictionaries written in STARDDL[7] a precursor version to DDLm.[2]

IDEALIZED MODEL

To introduce what we consider an idealized model for the relational language, we provide two early examples of its application. The first is a relatively simple relationship for the angle subtended by three bonded atoms, and the second shows how a complex mathematical relationship is expressed using dREL.

Received: February 7, 2012
Published: June 22, 2012

In Figure 1, the interatomic vectors $\mathbf{v}_{21}$ and $\mathbf{v}_{23}$ subtend the angle ϕ that may be calculated as its cosine from the inner




```
For [label] in _geom_angle.id
  {
    xc ++= _Cartn_transform.matrix * _atom_site[label].fract_xyz
  }

  v21, v23  =  xc[0]-xc[1], xc[2]-xc[1]

  _geom_angle.value =  Acosd( v21*v23 / (Norm(v21)*Norm(v23)))
```

Figure 1. Molecular diagram with the angle ϕ subtended by two interatomic vectors $\mathbf{v}_{21}$ and $\mathbf{v}_{23}$ and the associated dREL expression for evaluating ϕ.

product of the normalized vectors. This is clearly illustrated in the last line of the appended dREL script. Note that in this example the atomic vectors are first transformed from fractional positional coordinates into Cartesian coordinates in Ångstroms.

The second more complex dREL application is used in the scientific domain of crystallography. The crystal structure factor $F(\boldsymbol{h})$ for each reflection vector $\boldsymbol{h}$ has a complex value derived from the Fourier transform of the atomic structure in a crystal. The formula

$$F(\mathbf{h}) = \sum_{a}^{\text{atoms}} f_a(h) \sum_{s}^{\text{symmetry}} e^{-\mathbf{h}R_s\beta R_s\mathbf{h}} e^{2\pi i\mathbf{h}(\mathbf{R}_s+\mathbf{T}_s)\mathbf{x}_a}$$

is a summation over all independent atoms in the crystal asymmetric unit and over the symmetry equivalent units in the crystal cell. The dREL methods script expressing this relationship is shown in Figure 2.

The reader is not expected to understand the details of the formula in Figure 2, nor at this stage the syntax of the dREL

```
With h as refln

Loop a as atom_site {

 f = h.form_factor[a.type_symbol] * a.symm_multiplicity *
       a.occupancy

 Loop s as symmetry_equiv {

  _refn.F_complex += f *
          Exp(-h.hkl * s.R * a.matrix_beta * s.R * h.hkl) *
          ExpImag(TwoPi * (h.hkl *(s.R * a.fract_xyz + s.T)))
 }
}
```

Figure 2. Sample dREL definition for the structure factor for a reflection $\boldsymbol{h} = (h, k, l)$.

language (this is explained in more detail at the end of the paper), but the general correspondence between the formulas in these two examples and their corresponding dREL scripts is apparent.

First, the two dREL examples shown are formulaic and relatively simple to read. The items referred to in the scripts are defined in the same dictionary and will probably have definitions that themselves contain method scripts. Second, the methods scripts are devoid of bookkeeping complexities, such as specific data structures, list indexing, and the specifics of mathematical operations. Most importantly, those with knowledge of these relationships will quickly appreciate the roles of the terms in the formulas and should be able to construct them quite independently of their programming abilities. Last, from the canonical form of the script, the method can be machine-parsed and permits a *reverse* translation into the typographical formulas shown.

The dREL example in Figure 2 represents much more than an idealized model. It is the actual method script definition of the crystal structure factor, and as presented, it can be executed with the prototype dREL engine (described below) to determine values for each reflection $\boldsymbol{h}$. It is relevant to mention that an equivalent structure factor algorithm implemented in a conventional programming language, such as *Fortran* or *C*, would require several hundred lines of code.

The succinct form of the relationship information expressed in this example indicates the efficiency, clarity, and brevity that dREL algorithms are capable of. It is important to emphasize here, however, that the dREL methods and their execution are not intended to replace mainstream approaches to computation. Standard computer programs usually involve highly optimized code; implemented using complex data structures and programming shortcuts to ensure high speed and throughput. dREL scripts serve an entirely different purpose. Whereas the code for standard programs is written for experts, the dREL expressions are precise canonical descriptions of data relationships that are to be read by the wider domain community.

In this role, dREL supports conventional programming practices by providing a critical need to validate mainline program algorithms against a formal definition. The dREL methods, like those shown in Figures 1 and 2, provide important *reference implementations*.

DESIGN CONSIDERATIONS

Language Structure. In 1977, John Backus wrote a critique[8] of the conventional imperative programming languages based on the von Neumann architecture consisting of a CPU, memory for storage, with a communication bus to transmit a single word at a time between them. The bus is considered the bottleneck of these systems with most of the von Neumann cycle transmitting the what, where, and how of an operation rather than the doing. As a consequence, conventional programming languages are complex, and their semantics counterintuitive, because they use variables to imitate storage; control statements to affect jump and test instructions; and assignment statements imitate fetching, storing, and arithmetic. A programmer needs to include many bookkeeping steps and choreograph the multiple assignment and nested control statements, etc.

Several languages attempt to minimize programming overheads. The Structured Query Language[9] (SQL) used in relational databases is such a case. SQL supports a basic set

of relevant operations, *selection*, *projection*, and *Cartesian Product* and its store is entirely based on the concept of a *relation*. It deals with the abstract notion of a relation, and this enables a nonprogrammer to design relatively sophisticated queries over a relational database.

dREL is a *fit-for-purpose* language designed specifically to express data relationships in a dictionary definition written in DDLm.[2] The semantics of dREL are tailored to the data model of the STAR file[5] and to enable relatively complex relationships to be defined in a concise and transparent manner. The language expresses data relationships formulaically in a canonical form so they are easy to understand and to write. Concomitant with these design objectives, a dREL script is not isolated or local to the data item definition; it is fully dependent on, and utilizes information in, related items defined within the same dictionary. That is, each script is considered an integral part of a single data item's definition but at the same time can depend on the metadata of all related definitions within the dictionary. Another important design consideration is that dREL scripts be executable for specific data instances.

Data References in dREL. Syntactically, reference to a data value in dREL is achieved as follows.

1. A data item is most commonly referenced by its *internal* data name, constructed from a concatenation of the value of the attribute `name.category_id`, <*category*>, and the value of the attribute `name.object_id`, <*object*>, as <*category*>.<*object*>.
2. Reference to all of the multiple values of items in a *Loop* category is achieved by referring to the <*category*> component; this automatically provides the primary keys to every packet in a loop list.
3. Where a value (*fkey*) in one Loop category *A* represents a foreign key (a data value that is the primary key) to a different Loop category *B*, the construct <*B*>[*A.fkey*].<*object*> accesses the value <*B*>.<*object*> for the specific loop packet identified by *A.fkey*.
4. For a data item defined as type *List*, *Array*, or *Matrix* individual values are referenced by <*category*>.<*object*>[*n,m,...*] where *n, m,...* are the numerical indices to elements in that item.
5. For a data item defined as type Table an individual value is referenced by <*category*>.<*object*>[*label*] where *label* is the tag specific to the element.

Types of dREL scripts. dREL scripts are concise formulaic definitions of the target item in terms of its dependent data. There may be three separate types of dREL scripts in a single definition written within a DDLm dictionary.[2] The *Evaluation* method defines data relationships for generating the value of a *derivative* data item. *Primitive* data items cannot be evaluated and therefore contain no evaluation methods. The *Definition* methods may be used to tailor the definition according to instance data in scope. The *Validation* methods are can be used to ensure a data value is consistent with other data items in the instance document.

In the prototype dictionary engine, Evaluation methods have been fully implemented, and several sample Definition methods have been successfully tested. The Validation methods are similar to Evaluation methods and in principle can be implemented in a fully working version of the dictionary engine as Boolean functions. The Validation method returns *true* only if all its tests return *true*. At the simplest level, one test could be to confirm that the instance value is equal to the value generated by the Evaluation method.

Scope of dREL Scripts. An essential design requirement of dREL is that only the values associated with the defined data item are affected when the script is executed; all other changes are local and temporary. That is, each script is interpreted in the strictest sense as a *function* and the value returned by the method is the only data change; there are no nonlocal side effects. Of course, depending on the execution pathway, the methods of related data items may be invoked and these methods will independently update their own values. It must be emphasized, however, that each data value update is the result of the execution of its own method and is never the side effect of executing the method of some other data item.

dREL Expressions. dREL scripts may only refer to data items defined within the same dictionary. The reference is made using the defined data name. The presence of a data name in an expression is an implied *getter*, and the associated value is returned. This value may exist in the instance data file being processed, or otherwise returned by the evaluation method for this data item. This recursive process continues until the root method is completed or fails because a required value is missing from the instance file and no definition method for the item exists. Primitive data items (i.e., `type.source` `Measured`, `Assigned`, and `Observed` data) cannot be derived and hence their definitions cannot contain evaluation methods.

dREL Error Propagation. In science, the interpretation of number derived from an experimental measurement is meaningless without an associated error estimate. DDLm supports the definition of data items of the `Measurand` type, in which both the value and its standard uncertainty (*su*) are recorded. For data item definitions of type `Measurand`, the values derived from its Evaluation method must also generate an associated *su* value. Usually the *su* is appended to the number in parentheses, as in <*number*>(<*su*>).

Alternatively a data item of type `Measurand` can have a *declared* partner data item of the same name but with the suffix `_su` appended (e.g., <*tag*>`_su`). This partner item provides a separate placeholder for the standard uncertainty value *su*. In such a case the definition for <*tag*>`_su` must have its own Evaluation method script for the determination of the standard uncertainty. If <*tag*>`_su` is not defined in the dictionary, the present dREL implementation *automatically* approximates the standard uncertainty value *su* via the *first-order* propagation of errors by differentiating the terms in the tag methods script. In this case, the value and its *su* are expressed in an instance document as <*value*>(<*su*>) rather than two separate data items.

dREL Context Environment. Execution of a dREL methods script is intuitive and concise because the *data context at run-time* (i.e., the identifiers referring to categories and category packets currently in scope) are implicit in the execution pathway. The dREL run-time environment, under which methods may be executed, handles these tasks automatically.

For single-valued data items this execution process is simple because the *context* is the single instance data value. That is, the data value required is referred to by its data name and is either present, or its method returns its value. In the latter case, the evaluation result is automatically added to the instance data.

The execution context for multiple instances of a data item is handled slightly differently. A data item with multiple values

must, by definition, be part of a Loop category and appears in a looped list with all other data items of that category. A unique key defined for that category identifies the list packets. The dREL relationships for such data items will be in the context of each list packet and the data values contained within. The definition must execute in this scope when it is invoked.

In conventional programming languages that data value, and the indexes to all related packet data items, must be known and maintained by the programmer. For dREL scripts, the packet keys, and the allowed elements of the packet, are available from the dictionary definitions. In effect, the key that uniquely identifies a loop packet is kept in scope as the value of the reference to the method call. Whether this execution is undertaken sequentially for each individual packet, or executed in parallel for every packet, is an implementation level choice. The data evaluations using either approach will be the same.

EXECUTION MODEL

The recursive nature of a dREL execution pathway may be illustrated using the definition example of *density*. Crystal density is defined to be the ratio of the mass of atoms in the crystal unit cell divided the volume of the unit cell bounded by the cell vectors *a*, *b*, and *c*.

$$d = \frac{m_{\text{cell}}}{v_{\text{cell}}} = \frac{\sum_j^{\text{atom types}} n_j m_j^a}{a\cdot(b \times c)}$$

The definition of the `crystal.density`, as expressed in DDLm attributes,[2] follows in Figure 3.

```
save_crystal.density
  _definition.id          '_crystal.density'
  _definition.update      2011-06-20
  _description.text
;
  Crystal density calculated from crystal unit
  cell and atomic content.
;
  _name.category_id         crystal
  _name.object_id           density
  _type.purpose             Measurand
  _type.source              Derived
  _type.container           Single
  _type.contents            Real
  _enumeration.range        0.0:
  _units.code               megagrams_per_metres_cubed
  loop_
    _method.purpose
    _method.expression
                            Evaluation
;
  _crystal.density = 1.6605 *
                     _cell.atomic_mass / _cell.volume
;
save_
```

Figure 3. Definition of crystal density.

This `Evaluation` methods script states that the `crystal.density` value is the ratio of the `cell.atomic_mass` and the `cell.volume` values, scaled by the *amu* constant. Therefore if the `crystal.density` was not present in an instance data file, it could be evaluated directly from the values for `cell.atomic_mass` and the `cell.volume` provided these values existed in the file.

If `cell.atomic_mass` value is missing from the instance data file, its method, as given in Figure 4, will be invoked.

This method script needs to access data items with multiple values. The cell mass is a sum which depends on the number and type of atoms in the cell. In the instance file the values of these items reside in looped lists and individual values are accessed via a packet key. In the method below the `Loop` construct is executed over each key in the looped list, where the variable `t` refers to each category packet via its key, and `t.number_in_cell` refers to the `number_in_cell` item in this packet. Whether this is implemented as a sequential iteration, or in parallel for all packets, is not mandated in this specification.

```
save_cell.atomic_mass
  _definition.id          '_cell.atomic_mass'
  _definition.update      2011-06-20
  _description.text
;
  Atomic mass of the contents of the unit cell.
  This calculated from the atom sites present in
  the ATOM_TYPE list, rather than the ATOM_SITE
  lists of atoms in the refined model.
;
  _name.category_id       cell
  _name.object_id         atomic_mass
  _type.purpose           Quantity
  _type.source            Derived
  _type.container         Single
  _type.contents          Real
  _enumeration.range      0.:
  _units.code             daltons
  loop_
    _method.purpose
    _method.expression
                          Evaluation
;
  Loop t as atom_type  {
    _cell.atomic_mass += t.number_in_cell * t.atomic_mass
    }
;
save_
```

Figure 4. Definition of the cell atomic mass.

If the `atom_type.number_in_cell` is not present in the instance file, then the method in its definition shown in Figure 5 is invoked.

```
save_atom_type.number_in_cell
  _definition.id          '_atom_type.number_in_cell'
  _definition.update      2011-12-14
  _description.text
;
  Total number of atoms of this atom type in the unit cell.
;
  _name.category_id       atom_type
  _name.object_id         number_in_cell
  _type.purpose           Quantity
  _type.source            Derived
  _type.container         Single
  _type.contents          Real
  _enumeration.range      0:
  loop_
    _method.purpose
    _method.expression
                          Evaluation
;
  With t as atom_type

  Loop a as atom_site  {
    if ( a.type_symbol == t.symbol ) {
       _atom_type.number_in_cell +=
            a.occupancy * a.symmetry_multiplicity
    }
  }
;
save_
```

Figure 5. Definition of the atom count in a unit crystal cell.

Note that dREL exploits the programmer's *intention* by maintaining the reference to the current packet key for the `atom_type` list to a newly called method (see Figure 5) if it is needed. In our dictionary model, the methods of a data item in a Loop category cannot be sensibly invoked in the absence the packet key already in scope. That packet key remains in scope for the called routines also.

We purport that the sequence of definitions and methods scripts for crystal density is an appropriate exemplar to support the claim that dREL provides a straightforward and intuitive approach for defining precise relationships in data dictionaries that is meaningful to nonprogrammers. For computer

programmers, a dREL definition is also capable of being translated into code that exploits the full expressive power of conventional languages. For such a translation, additional runtime code needs to be injected into the software.

It is this latter approach that we have used in developing a prototype dREL parser and engine that can be executed. This execution engine is described in a later section.

DEFINITION STRUCTURE

A dREL method expresses the relationship between the defined data item and other data items in the dictionary. This approach leads naturally to *single evaluation pathways* during execution of a method and its related methods. The data *values* applied in a method are either present in the data instance environment, or are evaluated in related methods. The process is recursive until the root evaluation is completed. If an item value in the evaluation pathway is not present in the instance document, and no method exists in its definition (as is the case for primitive data),[2] then the root evaluation will fail.

To illustrate the recursive nature of data relationships, we will use the two closely related data items shown in Figures 6 and 7.

```
save_cell_vector.a
  _definition.id          '_cell_vector.a'
  _definition.update      2011-06-24
  _description.text
;
  The cell vector along the x axis.
;
  _name.category_id       vector
  _name.object_id         a
  _type.purpose           Measurand
  _type.source            Derived
  _type.container         Matrix
  _type.contents          Real
  _type.dimension         [3]
  loop_
    _method.purpose
    _method.expression
                          Evaluation
;
  _cell_vector.a = _cell.orthogonal_matrix * [1,0,0]
;
save
```

Figure 6. Definition of the crystal unit cell vector ***a***.

```
save_cell.orthogonal_matrix
  _definition.id          '_cell.orthogonal_matrix'
  _definition.update      2011-06-24
    _description.text
;
  Orthogonal matrix of the crystal unit cell. Definition
  uses Rollet's axial assignments with cell vectors a,b,c
  aligned with orthogonal axes X,Y,Z so that c||Z and b in
  plane YZ.
;
  _name.category_id       cell
  _name.object_id         orthogonal_matrix
  _type.purpose           Measurand
  _type.source            Derived
  _type.container         Matrix
  _type.contents          Real
  _type.dimension         [3,3]
  loop_
    _method.purpose
    _method.expression
                          Evaluation
;
  With p  as  cell_length
  With q  as  cell_angle
  With r  as  cell_reciprocal_angle

  _cell.orthogonal_matrix =   [
  [p.a*Sind(q.beta)*Sind(r.gamma), 0, 0],
  [-p.a*Sind(q.beta)*Cosd(r.gamma), p.b*Sind(q.alpha), 0],
  [p.a*Cosd(q.beta), p.b*Cosd(q.alpha), p.c]]
;
save_
```

Figure 7. Definition of the unit cell orthogonal matrix.

The value of `cell_vector.a` is determined from `cell.orthogonal_matrix` by a vector operation. If `cell.orthogonal_matrix` is not present in the instance data file, it is evaluated from its method. That method will fail unless all of the relevant data from the categories `cell_length` and `cell_angle` are present. The data items in these two categories are considered *primitive* and have no evaluation methods. The values of `cell_reciprocal_angle` can be determined from their dREL methods.

In some circumstances, missing primitive data values may result in a self-referencing loop in the recursive call stack, but this is easily detected at runtime.

These examples show how the dREL methods *declare* data relationships in a simple canonical form that is not written with the *execution pathway* in mind. If multiple execution pathways were present in a definition, then this would require the same algorithms to be duplicated for different data items.

To be more explicit, if a data item **R** has a dependence on a data item **S**, which in turn can be derived from **T**, then the preferred definition approach is that the method for **R** is written solely in terms of **S**, rather than by any indirect relationship to **T**. That is, it is not written as a conditional check for the presence of a value for **S** with an alternative evaluation based **T** if the condition should fail.

In the Figure 6 and 7 definitions, `cell_vector.a` is formally defined as the projection of the `cell.orthogonal_matrix` in the [1,0,0] direction. In turn, the `cell.orthogonal_matrix` is formally constructed from primitive scalar data. Clearly one could write the first method to check if `cell.orthogonal_matrix` is not set and construct the `cell_vector.a` via an algorithm using the primitive scalar data. This defeats the purpose of methods being formal definitions and creates the situation where parts of algorithms must be repeated in different data items.

PROTOTYPE DREL ENGINE

A dictionary engine has been implemented as a prototype *proof-of-concept* model to be used with data dictionaries written in DDLm. The prototype engine has been applied successfully for an earlier version of the DDLm, known as STARDDL.[7] The dictionary engine based on this prototype, and functional for the full set of DDLm attributes, is currently being developed under the auspices of the International Union of Crystallography.[10] We discuss below the key aspects of the dREL engine implementation based on experiences with our prototype.

Basic Requirements. At the simplest level, the dictionary engine cross checks the characteristics of each data value in an instance document against in the settings specified for each item with DDLm attributes.

For example, with the definition in Figure 8, the dictionary engine will verify that for every occurrence the data value of `atom_site.fract_xyz` exists a three-element vector (i.e., `Matrix` of dimension 3) of `Real Measurand` values (i.e., with associated standard uncertainty values). There can be multiple values of this data item because its parent category definition (`ATOM_SITE`) is a `Loop` category.

If the value of the vector `atom_site.fract_xyz` is not present in the instance document and the values of its separate scalar elements `atom_site.fract_x`, `*_y`, and `*_z` are, the dictionary engine will automatically use the relationship shown in Figure 9 to assemble the value of this vector.

Implementation Models. The two most common approaches to implementing a dictionary parser are illustrated

dx.doi.org/10.1021/ci300076w | J. Chem. Inf. Model. 2012, 52, 1917−1925

```
save_atom_site.fract_xyz
  _definition.id          '_atom_site.fract_xyz'
... [material deleted]
  _name.category_id       site
  _name.object_id         fract_xyz
  _type.purpose           Measurand
  _type.source            Assembled
  _type.container         Matrix
  _type.contents          Real
  _type.dimension         [3]
... [material deleted]
```

Figure 8. Definition attributes for a fractional position vector.

```
loop_
  _method.purpose
  _method.expression
           Evaluation
;
  With a  as  atom_site
  atom_site.fract_xyz = List([a.fract_x,
                              a.fract_y,
                              a.fract_z])
;
```

Figure 9. dREL method script of `atom_site.fract_xyz`.

in Figure 10. The first uses a generic parser that loads a dictionary and executes an interpreter over definitions including

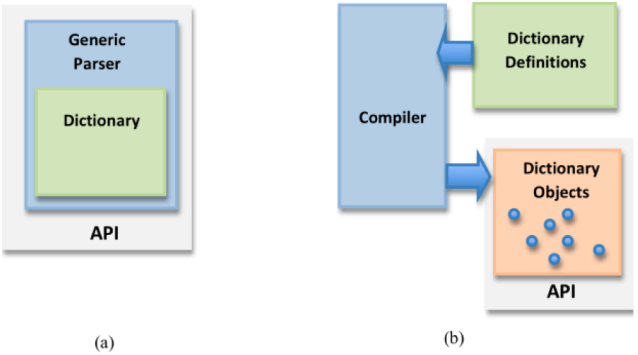


Figure 10. Different types of implementation models: (a) the interpreted dictionary approach and (b) the compiled dictionary approach.

the dREL scripts. The scripts are interpreted because the dREL is stored as source code, and its statements are executed as needed. This approach has the advantage that one parser is implemented, and any appropriately written domain dictionary can be added. The parser is in effect the API for any user-defined application to communicate their data requests through. Hester[11] has implemented a prototype parser using this approach.

A second approach uses a compiler that converts a dictionary directly into executable objects that become the sole source of definition information. This is the approach taken in our prototype implementation. By compiling the category and data item definitions into independent and intercommunicating runtime objects, the API can be constructed as a *class library*. Each runtime object is a self-contained definition that includes its current instance value and the dREL methods that may be executed over the object.

Compilation Models. There are several approaches to compilation. The traditional approach is to develop a compiler dedicated to a specific language that will generate executable machine code directly. This approach is efficient for programming languages in wide use because the considerable effort in building the compiler has a commensurate return. This was not a suitable development model for the prototype dREL engine.

The alternative approach used for the prototype engine translates the dREL scripts into an intermediate language and uses an existing compiler to generate any target machine code. This is an efficient and logical approach to building a proof-of-concept prototype. Moreover, the majority of data definition information is easily translated into an intermediate language because syntactic checking and simple validation are the dominant functions and this type of translation is common to all data items.

The translation of the canonical dREL scripts into an intermediate language involves considerable expansion of code. It is in this step, which is hidden from the dictionary user, that the true complexity of the data relationship is realized. In the prototype there are several rewrite rules that map the dREL specific syntax into its more generalized form in the target language.

Target Language. The dREL language specifications are summarized below and are described in full in the Supporting Information. dREL has a close relationship to the *Python*[12] programming language. The syntax, semantics, and constructors for strings, lists, and tables are the same, as are the assignment and operator overload semantics. There are several reasons for choosing Python. It is a relatively modern, rapid prototyping language for which extensions are available from a large open-source support community. It is a dynamically typed object-oriented language and possesses functionality that can be exploited by this work. Parser and compiler constructors that could be tailored to the needs of this project are readily available.

Python is also the intermediate language to which dREL is translated. It is also the implementation language of the dictionary parser. The choice of Python is not critical to an dREL implementation; it is simply convenient. A Python interpreter written in *Java*,[13] known as *Jython*[14] provides the facility to translate Python byte-code directly into Java classes. Additional user-defined Java classes generated from Java source code can be added, and this makes it possible to seamlessly blend programs written in Python and Java. Below we give an example application built with a user-interface written in Java's AWT that communicates directly with the dictionary parser and dREL engine written in Python and translated to Java classes.

Example Translation. Much of the simplicity in defining data relationships in dREL is achieved through the simple DDLm syntax. The first phase of compiling a dictionary is to translate this into the target language. This translation language is of course hidden from the dictionary user, but it is useful to provide an example here to illustrate how the overall implementation process works internally. The data item attributes, written in DDLm, are parsed and implemented as Python objects. The dREL scripts are parsed and translated via rewrite rules into pure Python code. This parsing sequence makes it possible, if desired, to include pure Python code within a dREL definition by letting it pass through the translator unchanged.

The translation into Python of the dREL given in Figure 9 is shown in Figure 11. The methods script in Figure 9 relates the

position vector atom_site.fract_xyz for an atomic site to its scalar x, y, and z positional components.

```
# Definition:  atom_site.fract_xyz

  def fract_xyz(_atom_site):  ①

  if hasattr(_atom_site,'fract_xyz_val'):  ②
    return _atom_site.fract_xyz_val;

  _atom_site.Computing('_atom_site','fract_xyz'); ③
  a = _atom_site;

  _atom_site.set_fract_xyz(List([a.fract_x(),  ④
                                 a.fract_y(),
                                 a.fract_z()]));
  return _atom_site.fract_xyz_val;

  def set_fract_xyz(self,v):
  self.fract_xyz_val = v       ⑤
```

Figure 11. Translation of dREL to an intermediate language.

The definition at ① is the *getter* method for the fract_xyz item for an object reference atom_site. The reference is the key to the packet of the atom_site loop list. The conditional statement at ② returns the value if the reference has the attribute value set. Otherwise ③ establishes the computation of this value from the code. In ④ the individual getter methods for the components fract_x, fract_y, and fract_z are called and a 3-vector is constructed from the returned values. This forms the argument to the set_fract_xyz() that sets the value in the reference object and, then, returns the value to the calling method. The set_fract_xyz() method is defined in ⑤.

Application. When the dictionary has been translated and compiled into a set of addressable runtime objects, each capable of returning the value of a data item, any application can be built to exploit this functionality. For example, an application has been built that reads any instance data file containing data items that conform to the domain dictionary and validates these items. If a data item has an *unknown* value (recorded as a ? character in the instance file), this application automatically invokes the dREL engine to perform a method evaluation (provided such a script exists in the definition) and inserts the evaluation into the instance data file.

In Figure 12 we show an example input instance file. The missing data values are highlighted. Note that the expected values (tagged as COMPARE) have been added for comparison with the dREL evaluation.

The application output is given in Figure 13. Note that comparing the output (in Figure 13) to the input file (in Figure 12), several data items have been added. These values have resulted from the recursive invocations of methods needed to determine the values of cell.volume and of exptl_crystal.density_diffrn.

For example to determine the density the cell.atomic_mass is needed and was determined from its dREL definition since it was missing from the file. In the figure, we expand the results for the two data items flagged as unknown in the original data file, and the dREL generated values for the volume and density are identical to their comparative values. Furthermore, the values generated by the parser are returned as Measurand values, with their standard uncertainties determined from the standard uncertainties of the dependent data items.

```
#
#
#    saly: calc vol, atom mult, density and bonds
#
#
data_Florence

_chemical_formula.sum                   'C7 H6 O3'
_chemical_formula.weight                132.07

_symmetry.cell_setting                  monoclinic
_symmetry.space_group_name_H-M          P_1_21/a_1
_symmetry.space_group_name_Hall         -p_2yab

loop_
_symmetry_equiv.pos_as_xyz
  +x,+y,+z  1/2-x,1/2+y,-z  -x,-y,-z  1/2+x,1/2-y,+z

_cell_length.a                          11.520(12)
_cell_length.b                          11.210(11)
_cell_length.c                          4.920(5)
_cell_angle.alpha                       90.00000
_cell_angle.beta                        90.8331(5)
_cell_angle.gamma                       90.00000
_cell.volume_COMPARE                    635.3
_cell.volume                            ?
_cell.formula_units_Z                   4

_exptl_crystal.density_diffrn           ?
_exptl_crystal.density_diffrn_COMPARE   1.444
_exptl_crystal.density_meas             1.234
_diffrn_radiation.type                  'Cu K\a'
_diffrn_radiation_wavelength.value      1.5418
```

Figure 12. CIF as an input instance data file.

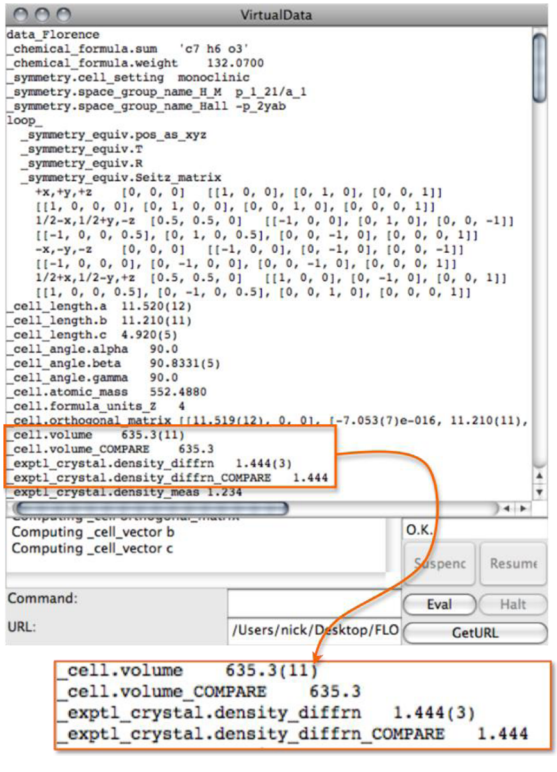


Figure 13. Automatic generation of missing data values.

Figure 14 shows the output of a dREL engine run in which values are calculated for an instance of data for the GEOM_BOND category of data. In this output, the true bond distances, with their standard uncertainties, are recorded as geom_bond.distance_COMPARE and the geo-

dx.doi.org/10.1021/ci300076w | J. Chem. Inf. Model. 2012, 52, 1917−1925

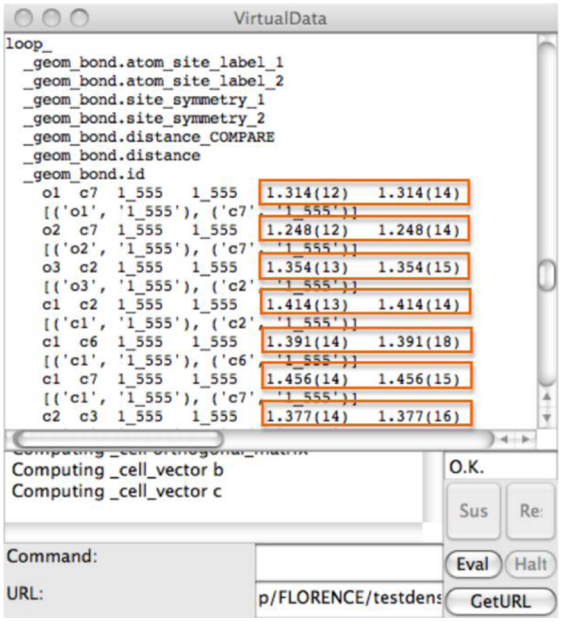


Figure 14. Automatic evaluation of bond lengths.

`m_bond.distance` values have been derived from the molecular structure data present in the instance file. Note the execution of the dREL definitions for this data item results in identical values with corresponding estimates of the standard uncertainties.

LANGUAGE SPECIFICATIONS

A complete description of the dREL language specifications is provided as Supporting Information. We provide here a brief overview of the main language specifications to assist the reader with the methods script examples provided earlier.

Language Basics. A named object or "variable" in dREL may only be created on assignment. The *typing* of a variable is by *coercion*. The *scope* of a variable is *local*.

Data Types. dREL supports all of the DDLm primitive data types. Local variable names are restricted to alphanumeric characters.

Variable typing is achieved explicitly by the dictionary definition of the data item, or implicitly by an expression or function return. DDLm dictionary definitions specify data types using the TYPE attributes (see `type.contents`, `type.container`, `type.purpose`, `type.source`, `type.dimension`).

Container Types. dREL supports the container types for data items

List	*List data is bounded by square brackets* [ ]
Array	*Array data is bounded by square brackets* [ ]
Matrix	*Matrix data is bounded by square brackets* [ ]
Table	*Table data is bounded by curly brackets* { }
Single	

Assignment Statements. dREL supports the following object assignments.

1. *transfer* is initiated with the = character in the following general form.

$$Lobjects = Robjects \text{ or}$$

$$Lobjects = \langle multiline\ expression \rangle$$

2. *incrementation* is initiated with the += digraph in the following general.

```
x += 2          i.e. initial value of x is 5, becomes 7
vect += 1       i.e. [3,3,3], becomes [4,4,4]
vect += [1,2,3] i.e. [3,3,3], becomes [4,5,6]
```

3 *decrementation* is initiated with the −= digraph in the following general form.

```
x -= 2          i.e. initial value of x is 5, becomes 3
```

4 *appending* is initiated with the ++= trigraph in the following general form.

```
vect ++= 1          i.e. [3,3,3], becomes [3,3,3,1]
vec2 ++= [1,2,3]    i.e. [3,3,3], becomes [3,3,3,1,2,3]
matx ++= [[1,2,3]]  i.e. [3,2,1], becomes [3,2,1,[1,2,3]]
```

Assignment Typing. In a dREL script, object types need not be declared. The typing of *Robjects* items may be determined from dictionary definitions, inline code constructions, or simply inferred by association with objects of known type. The TYPE of *Lobjects* may be set by the same mechanisms or result directly from the inferred type of the *Robjects* value.

It follows that the statement

```
x = 5
```

sets the TYPE of "x" as Integer. A new assignment of "x" in the next statement

```
x = 10
```

is permitted because it has a consistent TYPE. However, the new assignment

```
x = "Hello World"
```

is illegal.

Type Coercion Rules. Type coercion rules are needed when *Robjects* expressions contain objects of mixed type. dREL uses the following coercion rule, in order of increasing priority.

```
Integer → Real → Complex
```

Inline Comments. Inline comments are nonexecutable strings. In dREL, a sequence of characters following an unquoted *sharp* or *hash* symbol # is interpreted as a comment, up to the end-of-line character. Here are typical examples.

```
x = 5  # a comment follows an in-line hash
```

The following statement does not contain a comment because the hash symbol is contained within a quoted string.

```
s = "# this is *not* a comment"
```

Expression Operators and Terminators. The operands apply to *Integer*, *Real*, and *Complex* number objects. They are also applicable to the containers *List* and *Table* provided the elements of these are of TYPE *number*. The expression operators + and * have meaning for manipulating character strings.

Arithmetic expression operators

+	*addition*
*	*product (dot product when applied to vectors)*
^	*cross product of vectors*
**	*power of*
−	*subtraction*
/	*division*

Example statements using these terminators follow.

Logical expression operators

==	*equals*
!=	*not equals*
>	*greater than*
<	*less than*
>=	*greater than or equals*
<=	*less than or equals*
and	*and*
or	*or*
not	*not*
in	*matches element of the list*
not in	*does not matches element of the list*

Expression terminators

;	*semicolon*	separates multiple expressions in a line
\n	*newline*	closes a line unless unbalanced), } or]

```
a = 234 ;  y = 45 ;  z = -2
b = (y + z)/2.0
c = (45 + 72 *
      (93 + 4) + z)
```

Flow Control. dREL supports a range of standard and specialized flow control statements and terminators for controlling the repeated execution of object expressions. These are as follows:

```
Do
Repeat
For
Loop
With
Break
Next
If/ElseIf/Else
```

The essential constituents of a repetitive execution sequence are as follows.

```
<repeat-statement>   {
                     <expression block>
                     <repeat-terminator>(optional)
                     }
```

If more than one expression exists within the expression block, it MUST be enclosed within a set of braces { and }. If only one expression is repeated, its association with the *repeat-statement* is implied and the braces are optional.

SUMMARY

The dictionary methods language dREL is designed to enable relatively complex data relationships to be expressed as formulaic scripts in DDLm dictionaries. The simple but powerful canonical form of the dREL expressions is easy to read and understand and can be executed computationally. They expressions provide precise data dependency information to domain-specific definitions and are means for cross-validating data. dREL removes the programming barrier and encourages the production of the metadata needed for seamless data archiving and exchange in science.

ASSOCIATED CONTENT

Supporting Information

The complete specification of the dREL language is provided in the Supporting Information section. This material is available free of charge via the Internet at http://pubs.acs.org.

AUTHOR INFORMATION

Corresponding Author

*E-mail: Nick.Spadaccini@uwa.edu.au (N.S.); Sydney.Hall@uwa.edu.au (S.R.H.).

Notes

The authors declare no competing financial interest.

ACKNOWLEDGMENTS

We wish to thank Brian McMahon and James Hester for their feedback and advice during the preparation of this publication. We gratefully acknowledge the support of this work by the Australian Research Council Discovery Grant DP0344560.

REFERENCES

(1) Data's Shameful Neglect *Nature* **2009**, *461*, 145.

(2) Spadaccini, N.; Hall, S. R. DDLm: A New Dictionary Definition Language. *J Chem. Inf. Model.* **2012**, DOI: 10.1021/ci300075z.

(3) van der List, Eric. *The W3C's Object-Oriented Descriptions for XML*; O'Reilly Media: Sebastopol, CA, 2002.

(4) Spadaccini, N.; Hall, S. R.; Castleden, I. R. Relational Expressions in STAR File Dictionaries. *J. Chem. Inf. Comput. Sci.* **2000**, *40*, 1289–1301.

(5) Spadaccini, N.; Hall, S. R. Extensions to the STAR File Syntax. *J Chem. Inf. Model.* **2012**, DOI: 10.1021/ci300074v.

(6) (a) e-IRG Data Management Task Force; Report on Data Management, 2009; http://www.e-irg (accessed Sep 27, 2011). (b) UK Research Data Service; The Data Imperative: Managing the UK's Research Data for Future Use, 2009. http://www.ukrds.ac.uk/resources/download/id/14 (accessed Sep 27, 2011).

(7) Spadaccini, N.; Castleden, I. R.; du Boulay, D.; Hall, S. R.; Westbrook, J. D.; Berman, H. StarDDL: Towards the Unification of Star Dictionaries. *Acta Crystallogr., Sect. A: Suppl.* **2002**, *A58*, C256.

(8) Backus, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Commun. ACM* **1978**, *21*, 613–641.

(9) Date, C. J. *SQL and Relational Theory*; O'Reilly Media: Sebastopol, CA, 2009.

(10) Hester, J. R. Private communication, 2011.

(11) Hester, J. R. Implementing DDLm: rewriting dREL algorithms into other languages. *Acta Crystallogr., Sect. A: Suppl.* **2008**, *A64*, C635–6.

(12) van Rossum, G. *Python Tutorial*; Technical Report CS-R9526; Centrum voor Wiskunde en Informatica (CWI): Amsterdam, The Netherlands, 1995.

(13) Gosling, J.; Joy, W; Guy Steele, G. *The Java Language Specification*, 1st ed.; Addison Wesley: Reading, MA, 1996.

(14) Padroni, S.; Rappin, N. *Jython Essentials*; O'Reilly Media: Sebastopol, CA, 2002.