Geosci. Model Dev., 2, 33–42, 2009
www.geosci-model-dev.net/2/33/2009/

Geoscientific
Model Development

Spud 1.0: generalising and automating the user interfaces of scientific computer models

D. A. Ham[1], **P. E. Farrell**[1], **G. J. Gorman**[1], **J. R. Maddison**[2], **C. R. Wilson**[1], **S. C. Kramer**[1], **J. Shipton**[2], **G. S. Collins**[1], **C. J. Cotter**[3], **and M. D. Piggott**[1]

[1]Department of Earth Science and Engineering, Imperial College, London, UK
[2]Atmospheric, Oceanic and Planetary Physics, Department of Physics, University of Oxford, UK
[3]Department of Aeronautics, Imperial College London, UK

Received: 30 June 2008 – Published in Geosci. Model Dev. Discuss.: 17 July 2008
Revised: 11 February 2009 – Accepted: 27 February 2009 – Published: 6 March 2009

Abstract. The interfaces by which users specify the scenarios to be simulated by scientific computer models are frequently primitive, under-documented and ad-hoc text files which make using the model in question difficult and error-prone and significantly increase the development cost of the model. In this paper, we present a model-independent system, Spud, which formalises the specification of model input formats in terms of formal grammars. This is combined with an automated graphical user interface which guides users to create valid model inputs based on the grammar provided, and a generic options reading module, libspud, which minimises the development cost of adding model options.

Together, this provides a user friendly, well documented, self validating user interface which is applicable to a wide range of scientific models and which minimises the developer input required to maintain and extend the model interface.

1 Introduction

Computer models have become an indispensable tool in many fields of science and engineering. As models have become increasingly sophisticated and complex, the number of input parameters which control a typical piece of simulation software has become very large indeed. For example, Fluidity[1] is a multi-physics flow simulation package which supports multiple fluids and multiple phases in flow regimes as diverse as oceanography, porous media and flow inside nuclear reactors: the model therefore has several hundred control parameters.

Frequently, especially in the case of software developed in the course of research, the input parameters are read from one or more plain text files and the input system is coded on an ad-hoc basis for that model. The addition of new options as the model is developed typically requires that more code be added to read in the additional options, making model development cumbersome and tempting developers to engage in poor practices such as option overloading and the hard-coding of parameters which can make the model difficult to understand, maintain and use. From the model user's perspective, editing options files with a text editor is error-prone even when the documentation is excellent, which is frequently far from the case. Where graphical or "wizard" interfaces are available, the requirement to keep them current in turn increases the model development workload and therefore retards development of the scientific capabilities of the model.

In this paper, we describe a problem description system which provides model developers with a mechanism for ameliorating or avoiding altogether the problems described above. The options which control a model are described in a machine readable specification known as a *schema*. From the schema, it is possible to automatically generate a graphical user interface (GUI) which model users can use to set the control parameters. The schema can also be used to automatically check the input file for errors and this information can even be used by the GUI to help the user produce valid options files. A generic library is then used to read the options file generated by the GUI into the model code and those options can then be accessed as needed from within the model without the need for model-specific option parsing. The schema is therefore the only part of the system which differs from model to model. From the model developer's perspective, this system offers a ready made front end which naturally grows with the development of the model and which minimises the amount of code which must be

Correspondence to: D. A. Ham
(david.ham@imperial.ac.uk)

[1]http://amcg.ese.ic.ac.uk/Fluidity

written to add new model options. From the model user's perspective, the system provides a self-documenting graphical interface with real-time syntax checking which actively assists the user in generating a valid options file.

The system presented here, Spud, was developed to produce an options file format and user interface for the Imperial College Ocean Model (ICOM); however it has been designed to be useful for the widest possible range of scientific models and, in particular, for geoscientific simulation software.

2 Problem description languages

The key concept on which the entire system we present here is based is that of the problem description language. By this we simply mean a formal language in which simulation problems are described. The consequences of this approach are perhaps best explained by contrasting a formal language from existing options file formats, and by contrasting problem description from existing data file formats.

2.1 Formal languages

The most common existing practice in the field of scientific modelling software is that model parameters are specified in a text file, often with some informal grammar of keyword followed by value or values. In the worst case, there is no documentation and the validity of an input file is discernible only by what the model will or will not accept. Even where a file format is well documented, a user will frequently have to consult a large body of documentation in order to determine the correct expression in the terms of the options file of the simulation which is to be conducted.

Formal languages are a well understood concept in computer science and mathematics (see, for example Harrison, 1978). A formal language is defined by a vocabulary of symbols and a formal grammar which mechanically states which combinations of the vocabulary constitute well formed statements in the language. If the formal grammar is itself specified in a manner amenable to parsing by a computer program, it is possible for programs to automatically determine the validity of a statement and to ascertain the locations at which symbols can be added to a valid statement and still yield a valid statement. Among other things, the formal grammar determines which statements are required or optional, whether and how many times statements may be repeated and whether statements are required to be present in a particular order. As an illustration of the concept of a formal language, we may anticipate Sect. 4 and introduce a trivial formal grammar, or *schema* written in the Relax NG syntax:

```
start = (
    element a {
        element b {
            string
        },
        element c {
            xsd:integer+
        }?,
        element d {
            xsd:float
        }*
    }
)
```

Relax NG is a schema language for XML documents so this grammar defines an XML language. The schema above can be translated into English as saying:

"there will be an XML element *a* containing:

- an element *b* containing any string; followed by
- an optional ("?") element *c* containing one or more ("+") integers; followed by
- zero or more ("*") elements *d* each containing a single floating point number."

XML elements are delimited by tags consisting of the element name in angle brackets at the start of the element (`<a>`) and the same with the element name preceded by a slash at the end of the element (`</a>`). For example, the following statement is valid in the language defined by this grammar:

```
<a><b>test</b><c>1 2 3</c><d>10.0</d>
   <d>-5.0</d></a>
```

as is the much shorter:

```
<a><b>test</b></a>
```

since neither the *c* nor the *d* elements are required. However:

```
<a><b>test</b><d>3.0</d><c>red</c></a>
```

is invalid both because *c* elements can only contain integers and because a *d* is not permitted to precede a *c* element. It should be noted that this brief example does not reveal the full flexibility of formal languages and that, in particular, it is possible to permit much more flexibility in the ordering and content of elements than is presented here.

By formulating the combinations of options which constitute valid model input as a formal grammar, it is possible to mechanically determine the validity of model inputs. Conversely, it is also possible to write generic tools which, given a formal grammar, will guide a user to produce an options file which constitutes valid model input. The input file documentation can be interspersed with the specification of the formal grammar. This reduces the burden of documenting the options file format for the programmer and enables the user to be prompted with the correct documentation by the generic interface already mentioned.

An additional benefit of a machine parseable grammar is that the model itself can parse the grammar. This means that the library which reads the options file can be a generic tool which does not need to be modified as new options are added, or even for use with a completely different model in a different field of science.

2.2 Problem description versus data description

The problem of storing and handling the large volumes of data used and produced by scientific software is not a new one. In particular, large amounts of effort have been expended in defining standard file formats in which scientific data may be efficiently stored. As a leading example, many ocean and atmosphere models utilise the NetCDF data format (Rew et al., 2006), often in combination with the NetCDF Climate and Forecast (CF) Metadata Conventions (Eaton et al., 2008). The key distinction between these file formats and a problem description language is that the former is focussed on describing data, usually in a model independent manner, while the latter controls how a model deals with and produces data.

As a concrete example, consider the simulation of the flow in the North Atlantic with an ocean model. The initial conditions for velocity, temperature and salinity as well as climatographic data such as wind fields and temperature and salinity fluxes may be specified in a model independent manner in standard data file formats. However, in order to fully specify the action of the model on that data, it will also be necessary to prescribe model and problem specific parameters additional to the data. Examples of such data may include the time step and implicitness parameters, choices of parameterisations of viscosity and drag, convergence criteria for solvers, and instructions as to which of the data is to be applied as initial and/or boundary conditions.

It is therefore apparent that a problem description language is not an attempt to re-invent the standards for data file formats but rather is necessary to supply model and simulation specific parameters for use with model and simulation independent data.

2.3 A generic problem description system

In many respects the "holy grail" of problem description would be a generic language for some field of simulation, ocean circulation, say, which would formally describe a simulation to be performed including specifying the fields to be loaded from some standardised data file. This options file would then be readable by any model in the field which would then run the simulation and output standardised files ready for user intercomparison.

Advantageous though such a system would be from a user perspective, there are a number of reasons why this is not an achievable goal. To start with, even within one field, models have very different capabilities – ocean models may solve different flow equations, support different parameterisations and so on. A language which could drive all models would therefore either have to restrict itself to some minimal subset of functionality – in which case it would be of limited use to the user – or it would remain the case that an options file valid for one model would not be valid for another model – which defeats the purpose of a single language. From the developers' perspective, existing models may have strong assumptions about the structure and content of the input options and refactoring those to conform with an externally specified problem description language would be expensive and might be undesirable from the perspective of the scientific capabilities of the model.

Rather than attempting to define a generic problem description language, the system presented here, Spud, defines a framework in which model-specific problem description languages can be formulated and then provides generic tools which work with those languages. The design is based on separating those features of the problem description which can be considered universal to all scientific models from those which are specific to the model in question. The result is that for a model developer, there is an off-the-shelf problem description interface available. All that the developer need do is specify the grammar which defines which options the model takes and the user interface and parsing capability will be instantly available.

3 Components of the Spud system

There are three key components to the Spud system. The first is the mechanism for writing the formal grammar of a problem description language. This is based on XML and schemas written in RELAX NG and is documented in Sect. 4. The second component of Spud is Diamond. Diamond is a graphical user interface for writing options files in Spud languages. The schema written by the developer is used by Diamond to generate a model-specific interface which guides the model user to write a valid options file for that model. Diamond is documented in Sect. 5. The final component of the system is libspud. Libspud is a software library which will read any valid options file written in any Spud language into an in-memory representation. This enables direct access to any information in the options file from any point in the code with no requirement to pass option values through interfaces within the code or to add new parsing code as new options are added to the model. Section 6 documents this part of the code.

Figure 1 shows how these components fit together with a particular model. From the top left, the model-specific grammar, or *schema* defines the set of allowable options files using low level data components provided by the Spud base language. This schema is then used by Diamond to guide the

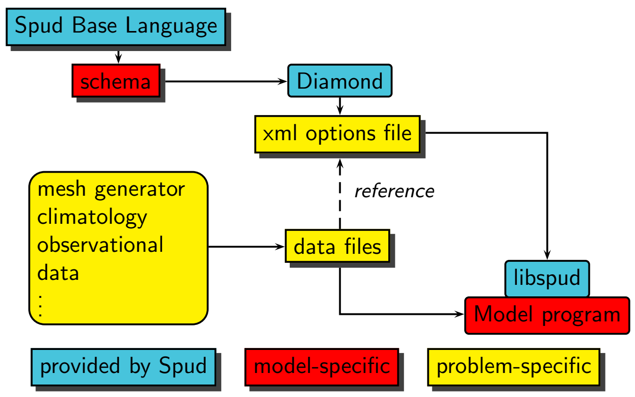


Fig. 1. Data flow in a scientific model using Spud. Blue components are supplied by Spud and are model independent, red components form a part of the model but are independent of the scenario being simulated and yellow components depend on the particular scenario.

user in generating an options file for the particular scenario to be simulated. The options file may, depending on the schema contents and the scenario in question, refer to external data sources to define some of the simulation inputs. Finally, the model is linked against libspud, which reads the options file. The references to external data in the options file may be used by the model to read the appropriate other data sources.

The full manual and source code of Spud 1.0 is included in the supplementary material (http://www.geosci-model-dev.net/2/33/2009/gmd-2-33-2009-supplement.zip).

4 Language specification in Spud

Unsurprisingly, the problem of specifying machine readable formal languages and formal grammars is a well studied one. In particular, the World Wide Web Consortium's Extensible Markup Language (XML) provides a generic syntax for machine parseable languages (Bray et al., 2006). XML files are organised as trees of nested elements. This is a particularly natural data model for a scientific model as it enables the relationships between options to be represented in the structure of the options file. For example, options to do with the time step can be grouped on one branch of the tree while options controlling, say, the discretisation of velocity can be grouped in another branch. The tree structure also provides a natural mechanism for sub-options: where a feature is selected and requires configuration, the options of that feature can be included as child elements of the main feature element.

In the XML system, the term *schema* is used to describe a formal grammar and there are a number of schema languages, where the term "schema language" refers to a formal language in which formal grammars for XML languages may be written. Some of the more prominent schema languages are analysed in Murata et al. (2005) while a less formal introduction is given in van der Vlist (2001). Spud utilises the RELAX NG schema language (Clark and Murata, 2001) which is a powerful schema language closely tied to the theory of formal tree languages (Clark, 2001). One of the important properties of a RELAX NG schema is that one schema can be imported into another schema. Spud utilises this capability to provide a set of rich data objects defining basic data types as building blocks for the schema developer. These core schema objects (known in RELAX NG as *patterns*) are described as the Spud base language. By specifying the low level representation of core data types, the generic tools included in Spud can handle low level data in a more elegant manner. The ability to include one schema in another also provides a mechanism by which coupled models could seamlessly utilise the Spud system.

4.1 Schema syntax: compact and XML forms

RELAX NG supports two completely different syntaxes for schemas. The compact syntax is optimised for human readability: the XML syntax is far more verbose but, being written in XML, is more readily supported by software parsers. The two syntaxes are absolutely equivalent in the grammars they express and it is possible to mechanically translate a schema between the two using the software package Trang[2]. Compact syntax is therefore the preferred syntax for editing Spud schemas and the Spud base language is shipped in this format. The complete schema is then translated to XML for use by Diamond. Figure 2 illustrates a fragment of a schema in compact syntax. This illustrates the representation of related options as nearby elements in a tree: there is one tree node which groups the time stepping options and specific time step options are children of this node.

4.2 Comments and documentation

A key advantage of the Spud system is that it enables documentation to be integrated with the schema. There are, in fact, three layers of documentation which are applicable, all of which are supported by the system. First, as with any programming language, RELAX NG permits comments which are used to document the markup of the schema for the benefit of schema developers. Much more importantly, Fig. 2 illustrates the use of RELAX NG annotations, marked by double comment markers. These annotations document the schema for the model users. The integration of user documentation with the schema has two benefits. First, it enables user tools such as Diamond to present that documentation to users as they formulate the options file. Second, they encourage the developer to write documentation at the same time as adding options to the schema. Since model manuals are notorious for lagging long behind model development, if they are written at all, this is a significant improvement. Relatively short comments associated with individual input

[2]http://www.thaiopensource.com/relaxng/trang.html

```
## Options dealing with time discretisation
element timestepping {
  ## Current simulation time. At the start of the simulation this is
  ## the start time.
  element current_time {
    real
  },
  ## Simulation time at which the simulation should end.
  element finish_time {
    real
  },
  ## The time step size. If adaptive time stepping is used then this
  ## is the initial time step size.
  element timestep {
    real
  }
}
```

Fig. 2. A fragment of compact RELAX NG schema defining some timestep parameters. This fragment defines the timestepping element and its child elements, current_time, finish_time and timestep. Real is a named pattern from the Spud base language for a scalar floating point value. The double comment marker ## marks user documentation which Diamond will present to the model user.

parameters are clearly not adequate as a sole source of documentation; however their presence significantly adds to the information readily available to the user and therefore contributes materially to the usability of the model in question. Finally, the Spud base language, which is discussed below, incorporates *comment* patterns which provide a mechanism for model users to comment their options files.

4.3 The Spud base language

There are certain core features which are common to a very large range of scientific software. By specifying certain schema features which will be common across all Spud languages, the possibility is created for Spud tools to deal with these basic model options in a common and powerful manner. Conversely the schema developer is relieved of the need to formulate low level schema representations of these basic features.

4.3.1 Problem dimension

It is a very common feature of scientific models, and geoscientific models in particular, that they simulate some physical region. The region may be three-dimensional but it is also common to model two or one dimensional regions. In some cases, higher dimensional domains may be modelled. While it is not uncommon that one software package may support modelling in domains of different dimension, it is generally the case that a given simulation will be conducted in a domain with a particular number of dimensions. That dimensionality will determine the number of components of vector options and the shape of rank 2 tensors. Vector and tensor valued solution fields are also likely to have a number of components determined by the overall problem dimension.

To facilitate tools which account for the dimensionality of the problem, Spud reserves the element *dimension* within a *geometry* element at the top level of the options tree to specify the problem dimension. The dimension specified here is used by the rich data types to ensure that numeric options conform to the problem dimension.

4.3.2 Rich data types

A large proportion of the options which are required to control simulation software are numeric quantities. Many of these are scalar real (floating point) values but integer quantities are also common, as are lists or vectors of values and indeed rank two arrays or tensors. It is also frequently necessary to supply short strings, such as file names for data sources, or much longer strings such as scripts for model-embedded scripting languages.

Spud supports numeric options by defining patterns for real and integer values of rank 0 (scalars), 1 (vectors) or 2 (tensors). Options with rank greater than 0 may either have unspecified extent, as is required when an arbitrary long list of values is to be supplied, or they may be given an extent related to the dimension of the problem, as might be required of a value such as a diffusivity. Tensor types may optionally be specified as restricted to symmetric tensor values.

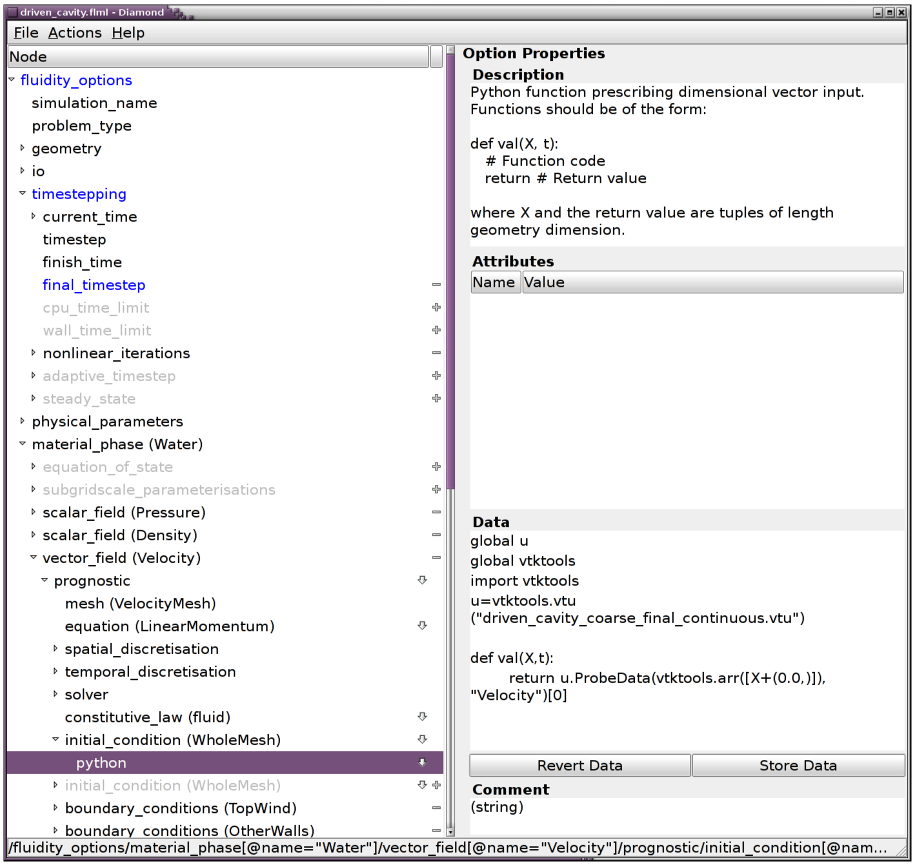
A basic *string* type which supports all strings is provided, as are *filename* and *python* types for file names and python functions respectively. The latter two facilitate additional validation of input by user tools. All rich data types have an embedded *comment* pattern to accommodate user comments.

4.3.3 Name attributes

Where a schema associates an attribute called *name* with an element, this has special significance in Spud. User tools are encouraged to display this attribute particularly prominently to enable users to distinguish between multiple identically named elements - for example multiple field elements in a flow model. Libspud allows model developers to access these multiple elements by specifying their name attribute. A name attribute must be specified in the schema for elements which are permitted or required to be repeated and may be specified for other elements. The value of the name attribute may be fixed in the schema or may be left as user-specifiable.

4.4 Cross-tree dependencies

As previously noted, XML languages are trees and the specifications of a schema language reflect this. This means that a schema defines which elements are permitted or required under which other elements. This can be used to specify quite sophisticated dependency relationships between model



Python function prescribing dimensional vector input.
Functions should be of the form:

```
def val(X, t):
   # Function code
   return # Return value
```

where X and the return value are tuples of length geometry dimension.

```
global u
global vtktools
import vtktools
u=vtktools.vtu
("driven_cavity_coarse_final_continuous.vtu")

def val(X,t):
        return u.ProbeData(vtktools.arr([X+(0.0,)]),
"Velocity")[0]
```

/fluidity_options/material_phase[@name="Water"]/vector_field[@name="Velocity"]/prognostic/initial_condition[@nam...

Fig. 3. A screen shot of the Diamond interface. The options tree, including greyed out unselected options and options with unset values (blue), is displayed on the left. The current option is the initial condition for the velocity field. On the right, the value of the current option is displayed in the space labelled "Data". In this case the option is an embedded Python script, specifying the initial velocity field from data in a vtu file. In the top right, the schema annotation for this option is visible while at the bottom right a space is available for user comments. Element names, provided by name attributes, are clearly displayed by elements such as fields.

options in a manner which is clear to the user. For example, a schema might permit the addition of any number of solution fields to a model and would then require that discretisation and solver options be added as a children of those fields. However, it is inevitable that there will be dependencies between options which are not related by a parent-child node relationship in the options tree. These dependencies are termed "cross-tree dependencies" and are not expressed, and therefore not enforced, by the schema. The absence of support for cross-tree dependencies is a key limitation of Spud. If it is desired that these dependencies are enforced, this can be achieved by the model interrogating the options tree provided by Spud and applying its own rules. The libspud interface which is introduced in Sect. 6 enables the model to interrogate the options tree but Spud does not itself provide a mechanism for specifying the dependencies themselves. A future version of Spud may incorporate a specific cross-tree dependency system based on a language such as Schematron[3].

5 Diamond

A very strong advantage of formally specifying the input language of a scientific model is that it facilitates the creation of universal tools. A tool can read a formal specification and work with the input language for a particular model without the need for case-by-case modifications.

A concrete example of this is Diamond which is a dynamic, language driven graphical user interface for creating validated input documents. Diamond parses an input RELAX NG schema, uses this to automatically generate a user interface, and allows the user to enter model configuration data. A screen shot of Diamond is shown in Fig. 3. Diamond is written in Python and uses the GTK+ toolkit[4] to build cross-platform graphical interfaces.

Whereas RELAX NG validators use the schema to decide the validity of a given document, Diamond instead uses the schema to determine what possible valid documents may exist. Therefore, the schema parser must be different to those

[3]http://www.schematron.com/

[4]http://gtk.org

found in RELAX NG validators. In fact, as it does not need to perform the derivation of regular expressions to compute the validity of a document (Brzozowski, 1964), it can be simplified significantly. Diamond uses the lxml XML library[5] to build an in-memory representation of the schema. Each element within this in-memory representation can be queried for valid child elements, and this information used by Diamond to create an interface with which to configure the elements; optional elements are greyed out with a button to activate them, elements that can be present multiple times have buttons to enable the addition of new instances or delete existing ones, and choices between different sub-trees are presented using a drop-down selection box. Similarly, information about element attributes and data are specified within the schema and stored in the in-memory representation generated by the parser. Diamond uses this to generate appropriate interfaces with which to edit the data. For example, with symmetric tensor data the user is presented with only those elements necessary to uniquely specify the tensor (the main diagonal and upper triangle).

Dynamically querying the schema in this way has two significant advantages. First, only valid input as specified by the schema may be created within the interface; the user may only generate elements in the correct numbers with the correct data types. This validity is enforced by the interface, relieving the user from mundane tasks such as the looking up of data types (e.g. integer or real), the length of vectors, or whether a tensor may be asymmetric. Diamond performs this validation dynamically, with elements whose data have not been set flagged to indicate to the user that they require attention. For example, in Fig. 3, the *final_timestep* element has been activated but its value not set. As a result it is coloured blue to mark its invalidity, as are its parent elements *timestepping* and *fluidity_options*. This validation can also detect if the schema has been changed, with newly added elements flagged as invalid until the user sets their attributes.

The second major advantage is that the interface does not need to be modified as the model input changes. Since the interface is generated automatically from the schema, any changes to the schema are themselves sufficient to change the interface. This allows for great flexibility in the model input, as only two tasks need be performed to add new model options: the addition of a new element to the schema, and the addition of code to the model to read in the new option. Due to the machine readability of the schema, any tools used in between these operations do not not require modification. This allows model developers to focus on the scientific functionality of their model and relieves them from the burden of maintaining an intuitive graphical interface for model users: the interface maintains itself.

[5]http://codespeak.net/lxml/

6 Libspud

Libspud is the software library which enables scientific models to access the options specified in an XML file written according to a Spud language. As with other components of the Spud system, libspud is model-independent. This means that libspud will work with any Spud language so that no changes need be made to the options reading mechanism as changes are made to the schema. Naturally it will still be necessary to modify the model code itself to make use of the new information contained in new options, but libspud reduces these modifications to a minimum. As noted in Sect. 4, XML files are trees of elements and libspud provides its generic interface into these trees by reading options files into an in-memory tree whose nodes correspond to the elements and attributes in the XML file. Options stored in this tree may then be accessed by specifying a string path similar to a reference to a file in a filesystem. This enables any option to be interrogated at any point in the model so long as its position in the tree is known. Interfaces to the options tree are provided in Fortran, C and C++.

At this stage it is important to re-emphasise the distinction between options and data which was raised in Sect. 2.2. The role of the libspud in-memory tree is to provide access to parameters specified in the options file such as the timestep, choice of numerical scheme and the location of files containing bulk input data. The storage of bulk data (such as solution fields and matrices) itself is not handled by Spud but is left to the existing mechanisms in the model. Indeed, it will generally be necessary to locally store options from the libspud tree in the local routines in which they are used. The advantage in this respect of the Spud system is that it removes the need to modify the options reading code and to pass the option in question through the call tree to the point at which it is used. Figure 4 illustrates the difference in the code changes which are needed to introduce a new model option. From this it can be seen that the addition of a new option is accomplished by adding its specification to the schema file and then accessing the option directly from the routine which implements the new feature. No additional values need to be propagated through the code either by function calls (as illustrated) or alternatively by some form of global, module or common variable arrangement.

6.1 Spud base language support in libspud

The named patterns supplied in the Spud base language produce rich data types for real and integer values of ranks 0, 1 and 2 as well as for strings. In the XML file these rich data types are represented by multiple elements but these are collapsed to a single node in the options tree. The type, rank and shape of these nodes can be queried and when the option value associated with the node is extracted it will be of the corresponding type, rank and shape. This facilitates options

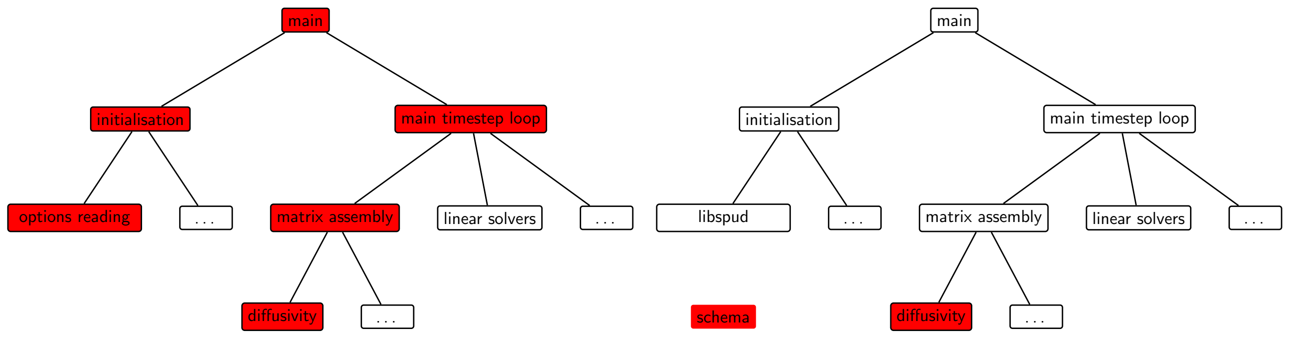


Fig. 4. Outline structure of a typical computational fluid dynamics package. The red backgrounds indicate the routines which must be altered in order to add a new diffusivity scheme. The left diagram indicates the conventional approach in which options are read by a custom routine and are passed as procedure arguments through the program to the point at which they are used. The right diagram shows the corresponding situation in a code using Spud: the new option is introduced in the schema and used in the diffusivity assembly routine. No other part of the program is touched.

which provide the values of, for example, diffusivity tensors or the value of a gravity vector.

6.2 Libspud in model code

From the developer's perspective, options are accessed as needed by referring to their location in the options tree. For example, suppose that the schema fragment shown in Fig. 2 occurs at the top level of the options tree. Then the model developer can retrieve the value of the model timestep with the Fortran call:

```
call get_option('/timestepping/timestep',dt)
```

where *dt* is a double precision real variable. In the schema used to drive the diamond interface in Fig. 3 there is an optional parameter which enables adaptive timestepping. Clearly the relevant routine in the model will need to test for the presence of this parameter. The following function call returns true if the parameter is present and false otherwise:

```
have_option('/timestepping/adaptive_timestep')
```

A more comprehensive example of the use of libspud in model code is presented in *ballistics.F90* in the *examples* directory in the accompanying source code while a full description of the entire libspud interface in Fortran, C and C++ is to be found in the manual (*doc/spud_manual.pdf* in the source directory).

7 The Fluidity Markup Language

Spud was developed to provide a new interface for the Imperial College Ocean Model (ICOM) (Pain et al., 2005; Piggott et al., 2008). ICOM is implemented as a part of a multiphysics finite element flow package called Fluidity and Spud has therefore been applied to Fluidity as a whole. The resulting problem description language is called the Fluidity Markup Language (FLML) and some details of its implementation are presented here as an example of the application of Spud. Figure 3 shows part of the FLML tree for a simple flow problem, the driven cavity problem (see Erturk et al., 2005). From this fragment, various aspects of the Fluidity problem description in FLML are apparent.

7.1 Field-centred options

The basic data object in any partial differential equation is the field: that is, a value associated with each point in the domain. For instance, the incompressible Navier-Stokes equations are solved for a vector valued velocity field and a scalar valued pressure field. There may also be scalar fields for quantities such as temperature and density and tensor valued fields for viscosity and diffusivities.

Fluidity supports a user specified number of scalar, vector and rank 2 tensor fields and many options are associated with individual fields. Individual fields are represented by *scalar_field*, *vector_field* or *tensor_field* elements and are differentiated by their *name* attribute. For example, an advected tracer will have temporal and spatial discretisation options and might even have additional fields associated with it to specify spatially and/or temporally variable quantities such as diffusivity or a source term. The expression of a partial differential equation problem in terms of the state fields is a prime example of the sense in which it is natural to represent problem options as a tree. Figure 3 shows pressure, density and velocity fields, with the velocity field expanded showing the next layers of options which apply to it. It can be seen that in this problem the velocity field is prognostic, as is usually the case. However to test other aspects of the model, the velocity can be switched to a prescribed field value for which no equation is solved.

Since, in this case, we are solving the incompressible Navier-Stokes equations for velocity, the field is specified as prognostic and this in turn switches on a subtree which contains all of the options which pertain to solving for this field. For instance, the *mesh* element determines which finite element space is to be used in representing this field while *solver* is a subtree containing all the options pertaining to the linear solver which will be used to solve the discretised system.

7.2 Support for multiple materials and phases

Fluidity supports simulations of multiphase flow, for example for simulating the flow of oil and water through porous rock (Saunders et al., 2006) and a capability for simulating multimaterial scenarios such as fluid-solid coupling and meteorite impacts is in development. This is supported in FLML by grouping fields into an arbitrary number of *material_phase* elements. A material phase groups together fields which share the same velocity, density and pressure. Material phases have their own equation of state. Figure 3 shows the deactivated *equation_of_state* element which is not required in this simulation as the driven cavity scenario has a prescribed constant density. This example illustrates the manner in which Diamond presents users with the options which are available, rather than relying on the user finding this information in whatever manual for the simulation software is available.

8 Applicability to other models

It has been emphasised throughout this paper that Spud is model-independent with only the schema varying between models. The size and complexity of the schema is directly related to the number of options supported and the complexity of their interdependency. In addition, the provision of the spud base language means that there is essentially no groundwork to be done in supporting basic option types. This makes Spud applicable to a wide range of models from the smallest projects undertaken by a single PhD student up to very large multiphysics packages with dozens of developers at multiple sites, as is illustrated by the example of Fluidity given above. Spud can also be retrofitted to a model at modest development cost. For instance, Spud has been applied to the FullWave seismic tomography model which has approximately 100 user parameters. This project was substantially completed with less than one person-week of effort.

9 Distribution and licencing

Spud and Diamond are available from the Applied Modelling and Computation Group at Imperial College London[6]. All components of the package are free software with Diamond being licensed under the GNU General Public License version 3.0 and the other components under the GNU Lesser General Public License version 2.1. This combination of licenses ensures that Spud can legally be used with models which employ a wide range of licensing schemes, both free and proprietary. Full details of the licenses, including the (compatible) copyright notices of some third party routines included in the package, are included in the *COPYING* and *diamond/COPYING* files in the source distribution.

[6]http://amcg.ese.ic.ac.uk/spud

10 Conclusions

Scientific computer models of ever increasing complexity are a cornerstone of modern science. The problem of specifying all of the options which control these models is one which presents a serious development cost for model developers and a significant barrier to users ability to set up simulations without making errors. Here we have presented a significant departure from the existing practice in geoscientific models. By specifying a formal grammar for an XML problem description language, we have developed generic tools for both writing option files and for accessing those options from within the model code. These tools guide the model user to set up a valid problem description, while from the developers' perspective, the cost of adding new options as the model is improved and expanded is minimised. The integration of documentation with the formal grammar further encourages developers to properly document the model and presents the users with documentation integrated in the GUI.

We have demonstrated the feasibility of this approach and illustrated its application to the Imperial College Ocean Model as a part of the general fluid mechanics package Fluidity. Spud is also being applied to the FullWave seismic tomography package and we are confident that the advantages of this approach will lead to its adoption in further models in the geosciences and beyond.

Acknowledgements. This work was supported by a NERC consortium grant (numbers NE/C52101X/1 and NE/C521028/2), NERC fellowships NE/C51829X/1 and NE/E013589/1, and NERC studentship NE/F007833/1.

The authors would like to thank Paul Kelly for useful discussions during the development of this work.

Edited by: I. Rutt

References

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cowan, J.: Extensible Markup Language (XML) 1.1 (Second Edition), Tech. rep., World Wide Web Consortium, http://www.w3.org/TR/xml11/, 2006.

Brzozowski, J. A.: Derivatives of Regular Expressions, J. ACM, 11, 481–494, doi:10.1145/321239.321249, 1964.

Clark, J.: The design of RELAX NG, http://www.thaiopensource.com/relaxng/design.html, 2001.

Clark, J. and Murata, M.: RELAX NG specification, Tech. rep., Organization for the Advancement of Structured Information Standards, http://relaxng.org/spec-20011203.html, 2001.

Eaton, B., Gregory, J., Drach, B., Taylor, K., Hankin, S., Caron, J., and Signell, R.: NetCDF Climate and Forecast (CF) Metadata Conventions, Version 1.1, http://cf-pcmdi.llnl.gov/documents/cf-conventions/1.1/cf-conventions.pdf, 2008.

Erturk, E., Corke, T., and Gokcol, C.: Numerical solutions of 2-D steady incompressible driven cavity ow at high Reynolds numbers, Int. J. Numer. Meth. Fl., 48, 747–774, doi:10.1002/fld.953, 2005.

Harrison, M. A.: Introduction to Formal Language Theory, Addison-Wesley Longman, Boston, MA, USA, 1978.

Murata, M., Lee, D., Mani, M., and Kawaguchi, K.: Taxonomy of XML schema languages using formal language theory, ACM Transactions on Internet Technology (TOIT), 5, 660–704, doi:10.1145/1111627.1111631, 2005.

Pain, C., Piggott, M., Goddard, A., Fang, F., Gorman, G., Marshall, D., Eaton, M., Power, P., and de Oliveira, C.: Three-dimensional unstructured mesh ocean modelling, Ocean Model., 10, 5–33, doi:10.1016/j.ocemod.2004.07.005, 2005.

Piggott, M., Gorman, G., Pain, C., Allison, P., Candy, A., Martin, B., and Wells, M.: A new computational framework for multiscale ocean modelling based on adapting unstructured meshes, Int. J. Numer. Meth. Fl., 56, 1003, doi:10.1002/fld.1663, 2008.

Rew, R., Hartnett, E. J., and Caron, J.: NetCDF-4: software implementing an enhanced data model for the geosciences, in: 22nd International Conference on Interactive Information Processing Systems for Meteorology, Oceanography, and Hydrology, http://ams.confex.com/ams/Annual2006/techprogram/paper_104931.htm, 2006.

Saunders, J., Jackson, M., and Pain, C.: A new numerical model of electrokinetic potential response during hydrocarbon recovery, Geophys. Res. Lett., 33, L15316, doi:10.1029/2006GL026835, 2006.

van der Vlist, E.: Comparing XML Schema Languages, XML.com, http://www.xml.com/pub/a/2001/12/12/schemacompare.html, 2001.