Geosci. Model Dev. Discuss., 1, S138–S145, 2009
www.geosci-model-dev-discuss.net/1/S138/2009/

Geoscientific
Model Development
Discussions

Interactive comment on "Spud 1.0: generalising and automating the user interfaces of scientific computer models" *by* D. A. Ham et al.

D. A. Ham et al.

Received and published: 19 January 2009

We would like to thank the reviewer for his constructive comments on our paper which have enabled us to produced a revised version which we feel is significantly improved.

Namelist files

We did not discuss namelist files in the paper, partly because they are specific to one programming language (Fortran) and would therefore add little clarity to those readers not familiar with that language, but mostly because namelist files are not sufficiently different from keyword value text files to warrant separate consideration in the paper. Happily, however, GMDD provides this alternative forum to publicly discuss detailed concerns and we are happy to do so here.

There are a number of respects in which namelist files fail to provide the facilities which

Full Screen / Esc
Printer-friendly Version
Interactive Discussion
Discussion Paper

are provided by Spud:

Lack of grammar Namelist files have a syntax defined in the Fortran standard but they lack any mechanism short of reading the source code of the model for providing either the user or an interface tool with the information about which options are permitted or required and what their ranks, shapes and types are. In particular, the lack of a formal grammar makes it essentially impossible to write a generic user interface for namelist files. The most important feature of the Spud system is that it improves model usability by supplying information to the user about valid options combinations. Namelists do not do this.

No support for dynamic sizing Namelist variables are explicitly prohibited by the Fortran standard from containing any dynamically sized or allocated variables. Take as an example a fluid dynamics code which allows the user to specify any number of tracer variables. It will be necessary for the input files to specify initial and boundary conditions for a variable number of fields. Since there must be a fixed number of namelist variables each of a fixed size, this is essentially beyond the capability of namelist input.

Limited support for grouping of options The reviewer points out namelists support the logical arrangement of options via namelist groups. While this is true, namelists support exactly one layer of grouping. Referring to the options tree presented in figure 2 of the GMDD paper it is apparent that, at least for more complex models, there are multiple layers of grouping which are appropriate and that using multiple layers of grouping and allowing these groups to be enabled and disabled as appropriate makes the model significantly more comprehensible to the user. It also makes it easier for the developer as options can be retrieved relative to other options in the tree which makes it very easy to determine, for instance, which discretisation options apply to the current field.

GMDD
1, S138–S145, 2009

Namelist files provide a convenient pre-defined mechanism for inputting keyword value option files. This is very convenient when producing a simple model with perhaps a dozen or so control parameters. As model complexity grows, namelists become unmanageable and, in comparison with Spud, detract from the usability of the model.

Formal grammars and languages

In the revised paper we have included the following example of formal language and grammar usage:

> 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

GMDD

1, S138–S145, 2009

Interactive Comment

- 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 (). 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.

Scripted modification of options files

Because of the standardised tree structure of XML files, it is particularly easy to uniquely address individual elements in a file and high level text handling programming languages such as Python include standard tools for accessing and modifying

GMDD
1, S138–S145, 2009

element values in XML files. The standard mechanism for accessing an element in an XML file is XPath and Spud directly supports simplified XPath addressing. Indeed, users need not even be familiar with XPath as the latest version of Diamond allows users to copy the path for the currently selected option to the clipboard using the "copy spud path" command in the edit menu.

The latest Spud source tree, included in the revised supplementary materials, contains the very short (19 executable lines) script spud-set which can be used to change the value of any option in a Spud options file given the path of the option to be changed. This makes it trivial to script modifications to input files to conduct multiple related simulations. Spud-set is documented in chapter 5 of the revised Spud Manual in the doc directory of the Spud sources.

Model performance

The reviewer states with respect to section 2.3:

> lines 11-22: I would argue that model performance is the most limiting factor, rather than the reasons stated here.

We fear there must be some misunderstanding here. The point of this paragraph is to convey the idea that it is not possible to write a single problem description language (i.e. an options file format) which could be used directly to conduct the same simulation using a number of different models. Model performance (of any sort: efficiency, accuracy or capability) is a completely different question. Indeed, the desire to run the same scenario on a number of different models is often motivated by the desire to learn something about model performance.

Examples

It is not really appropriate to provide complete examples in the journal text, however we have added a completely coded example called ballistics in the examples directory

GMDD

1, S138–S145, 2009

Interactive Comment

of the Spud sources in the revised supplementary materials. This includes a sample Fortran program (*ballistics.F90*) and schemas in compact (*ballistics.rnc*) and XML (*ballistics.rng*) formats.

In-memory representation

The reviewer writes:

> It is not clear to me what exactly is meant by "in-memory"; and how a model acesses the parameters specified in the input file. Is it as simple as simply having a variable, for example, r_pie, which can then be given a value in diamond, which is then be accessed in the model using the variable "r_pie" (where r_pie is defined by libspud)? It doesn't appear to work this way...

It very nearly works this way. Clearly the contents of the schema or input file cannot directly define variables in the model. The point of the in-memory representation is that libspud reads in the options file and stores the values of all the options in memory. These can then be queried at any point in the model, rather than having to pass those values through the code. We have added the following illustration of this to Sect. 6:

> 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. 1 occurs at the top level of the options tree. Then the model developer can retrieve the value of the model time step 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. 2 there is an optional parameter which

GMDD

1, S138–S145, 2009

enables adaptive time stepping. 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/manual.pdf* in the source directory).

Development cost

The reviewer writes:

> as it is currently done, a developer is only required to define a variable and read it in, rather than setup a series of additional routines on top of an entire XML frontend.

With respect, this massively underestimates the complexity of reading options and passing them through a complex geoscientific model. For example some options will only be present if certain discretisations are selected. Other options may vary in number: a field may have one boundary condition or its boundary may be subdivided into many parts each of which has different boundary condition options. Reading and storing options in memory is only a trivial task if the number of configurations of a model are trivial.

The advantage of Spud to the developer in adding options is that the dependencies, type, rank and shape of the option can be declared in a single neat and efficient manner in the schema and Spud make those values available anywhere in the model where

GMDD

1, S138–S145, 2009

they need to be used without the developer having to deal with the logic of determining which options to read and where to pass them. Our experience is that this makes it much, much easier for developers to add options and to do so in a way that improves model usability rather than detracting from it.

Interactive comment on Geosci. Model Dev. Discuss., 1, 125, 2008.

GMDD

1, S138–S145, 2009