Project Number GA: **228203**

Project Acronym: **IS-ENES**

Project Title: **InfraStructure for the European Network for Earth System Modelling**

Programme: **SEVENTH FRAMEWORK PROGRAMME**
Capacities Specific Programme
Research Infrastructures

D8.3 – Towards Flexible Construction of ESMs using BFG

WP8/JRA2: *European ESM: Performance Enhancement*

Due Date:		M36
Submission Date:		-
Start Date of Project:		01/03/2009
Duration of Project:		48 months
Organisation Responsible for the Deliverable:		UNIMAN (6)
Version:		1.0
Status		Final
Author(s):	R. W. Ford, G.D. Riley	UNIMAN (6)

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	01/08/2011	First draft: for comment	Rupert W. Ford, Graham D. Riley
0.2	10/01/12	Updated draft for internal review	Rupert W. Ford, Graham D. Riley
0.3	13/02/12	Updated draft for internal project review	Rupert W. Ford, Graham D. Riley
0.4	05/03/12	Updates following internal review	Rupert W. Ford, Graham D. Riley, Sophie Valcke
1	26/03/12	Final updates following iternal review	Rupert W. Ford, Graham D. Riley, Sophie Valcke, Joachim Biercamp

Table of Contents

1. Executive Summary

This work package undertakes research into the performance aspects of configuring, deploying and running Earth System Models (ESMs).The work package covers a number of key areas relating to model performance including: research portability and performance of key models on a range of platforms, including emerging petascale PrACE machines; work to develop tools to ease the composition of new ESMs from existing model components and coupler technologies which will help to lower the technical hurdle for small climate research organisations.

Within these objectives, task 3, Flexible Construction of ESMs, undertakes research into future coupling technologies seeking to provide flexibility in the construction and deployment of future, community-based, ESMs using appropriate underlying coupling systems, such as OASIS and ESMF.

This document is organized as follows; Section 2 provides a general introduction to the Bespoke Framework Generator version 2 (BFG2) Generative Programming system; Section 3 presents a new Scientific API which supports both of the prevalent coupling styles in use by the community today (termed aggregation and composition in this document); Section 4 discusses progress towards supporting framework interoperability (i.e. being able to couple models that are already written to conform to a specific framework); Section 5 briefly discusses language support that is being added to BFG2 to help integrate with other domains, for example, Integrated Assessment; Section 6 presents the prototype BFG portal, where one can run BFG and browse model, transformation and coupling descriptions; Section 7 describes progress towards the support for consistency between Metadata descriptions of model code and the code itself; Section 8 explains how and why the internals of BFG2 are being re-written to use templating, implemented in PYTHON, (replacing the use of the previous techonolgy, XSLT); in Section 9 two examples of the use of BFG2 are provided, the first showing interoperability with ESMF and OASIS, and the second in the domain of Integrated Assessment Modelling. A third example examines the issues involved in making a complex code compliant to a component-style interface; finally, Section 10 provides a general summary of the previous sections.

Further information and references to papers related to BFG can be found on the BFG website: http://www.cs.man.ac.uk/cnc/projects/bfg.

A BFG2 wiki page can be found at: https://source.ggy.bris.ac.uk/wiki/GENIE_BFG.

A useful reference to BFG2 is:

C.W. Armstrong, R. W. Ford and G. D. Riley. *Coupling integrated Earth System Model components with BFG2*, Concurrency and Computation: Practice and Experience, Vol. 21 No. 6, pp. 767--791, 2009, DOI: 10.1002/cpe.1348.

2. BFG2 Approach

2.1 Motivation

Developers and users of Earth System Modelling software employ (scientific) models in many different scenarios. For example, a model may initially be developed as a single column model on a laptop or workstation. Once debugged, the same model may be incorporated into a coupled model which is developed and debugged on a workstation or on a local cluster before, eventually, being deployed in production mode on one or more high performance computers. Typically, a model continues to be developed over time with potential consequences for the coupled models, and other scenarios, in which the model is used. A model developer may also be required to provide the model to a user in such a way that it conforms to the user's chosen coupling infrastructure, for example, as an ESMF-compliant model, or an OASIS-compliant model. At some point in its life, this model may be used in another scientific application domain, for example, in Integrated Assessment Modelling. Finally, a community coupled model, such as those in the CIAS system, which includes the original model, may be run in a distributed manner, in a cloud or as a web service (or even as a set of web services), possibly as part of a more general, formal scientific workflow process.

The software infrastructure used in these different scenarios will be very different. There is a good reason for this, the requirements of developers and users will be very different in each scenario, and software that is tailored to a user's requirements is typically better, both in terms of ease-of-use and performance, than more generic software. As a crude illustration, one would not expect to communicate coupling data using the web service transport language SOAP on a Supercomputer. However, although the software infrastructure is expected to be different depending on the deployment scenario, crucially, the underlying science code that is run, should remain the same in all cases. The effort to port the scientific model code to each of the software infrastructures, and support it as further developments take place, would be expected to be significant over the lifetime of the the model. Such porting is error prone and costly. The cost of reusing scientific model software will become a major problem in the future as more, and more diverse coupled modelling communities who wish to share and exchange their scientific models emerge.

This document describes an approach that provides a general, flexible, solution to the above problem. The proposal is that developers of Earth System Modelling scientific software adopt a standard, relatively low level interface for specifying the details of couplings etc. that separates science code from the specific details of and particular software infrastructure used to implement a couple model in code. This approach frees the model from being limited to using a particular software infrastructure and also isolates the model from future changes in infrastructure code.

There are currently two dominant styles of coupling in use in Earth System Modelling: in OASIS, models are written as programs which use in-place calls to provide for the exchange coupling data; in ESMF, models are written in a component-style with data sharing used to specify coupling data[1]. Current practice in the community is that a developer of a coupled model must choose between the two styles. However, there is agreement in the coupling community that supporting both styles *within a single coupling system* would be beneficial.

One of the conclusions from the IS-ENES-sponsored workshop on Coupling Technologies for ESM[2] in 2011, at which there were representatives of all the main ESM coupling systems, was the following:

[1]In the OAISIS infrastructure, the in-place calls map in a reasonably direct way to calls to MPI. With ESMF, the actual mechanism used to exchange coupling data depends on how the models are deployed and may involve shared data or message passing.

[2]http://pantar.cerfacs.fr/globc/publication/proceed/2011/Proceedings_of_the_workshop_final.pdf

"For maximum coupling flexibility and efficiency, all climate component models should be refactored into init, run and finalize units. Where the norm is a multiple executable approach, such as the European climate modelling community, it may be difficult to achieve the agreement on component interfaces required for integrated coupling. To satisfy all cases, an "ideal" coupling technology should therefore offer both approaches in order allow an easy assembling of legacy code but also provide more efficient and flexible coupling when interface agreements can be reached. Current research in Generative Programming explores approaches that may enable such an "ideal" coupling technology to be built."

In the IS-ENES project, BFG has been extended to allow models to be made compliant to either of the above two styles and, using BFG's code generation facility, models written in either style can be coupled together in any combination, and the coupled model can use the appropriate, user-specified coupling infrastructure.

In addition to writing BFG-compliant software, a developer must provide metadata describing: each model in a coupled model; how the models are to be coupled together (in terms of the coupling data to be exchanged); how the models are to be deployed onto appropriate computing resources. Deployment metadata also includes details of any specific coupling infrastructure, such as OASIS or ESMF, to be used to exchange coupling data.

BFG is a generative programming tool which takes in the metadata describing a coupled model and produces the appropriate 'wrapper' code required to build and execute the coupled model using the software infrastructure most suited to their particular requirements.

2.2 Science Interface

The Bespoke Framework Generator version 2 (BFG2) defines an interface, including some minimal coding rules, to which (scientific) model code must adhere if a model is to be used with the system. The aim of this interface is to provide a formal separation of the science code from the software infrastructure, including specific coupling software. This separation promotes the ease of re-use of the science code, in other modelling contexts, and insulates the code developer from changes in the APIs of software infrastructure. As such it is aimed at a level below current coupling API's such as OASIS and ESMF, for example, and can be used in conjunction with these, if required. BFG provides two APIs: the **component API** and the **program API**. These are described next.

The **component API** is intended to be used with highly modularised code and is designed to be as close to the way in which modular code would be naturally written as possible. In Fortran90, for example, a model may be written as a module with one or more public subroutines. Coupling data from other models (in a coupled model) can be passed to and from the subroutines using **argument passing**. In fact, this structure is already what some of the existing, more modular, coupling systems implement in an informal way, such as, for example, the models at GFDL[3].

Coupling data may also be passed to and from the subroutines of a model through the addition of in-place **put and get** calls in the code. The put and get calls require the user to provide a reference to the data that they wish to input or output (often the name of an array), and also to provide a tag, that is unique[4], for the data. The tag is used to link the data in a call to metadata descriptions that the model developer must also provide (BFG metadata is described in Section 2.4). The developer is able to choose whether they prefer to use

[3]In a private communication with Balaji

[4]Uniqueness is only required within the model and also within a put or a get.

arguments or put/get calls on a field-by-field basis.

A natural question would be why one would want to provide a put/get interface if an argument passing one exists. One answer to this question is that some data that is output by a model might only be short lived when the code executes: the data could be local to a deeply-nested subroutine, for example; forcing a developer to make this data global and pass it out via a long chain of argument lists would increase the memory footprint of the program and require the developer to change their code. It would also mean that the data was output at the end of the code, rather than from where it was created. A practical example of this is diagnostic data.

In addition to the component API, a **program API** has recently been added to BFG as many modelling groups, especially those new to coupled modelling, have existing scientific models that are implemented as programs. The program API allows whole programs to be made compliant for use within a BFG-managed coupled model without the need for modularising them first, as would be required if only the component API existed. In this case, one must specify the data to be input and output by the program through the use of explicit put and get calls. The program must also have some additional routines added to mark important phases of the program execution. The additional routines include calls to mark the start and end of the program and to mark the end of an iteration (in time-stepping or convergence loops, for example). The use of the program interface reduces the cost of integrating existing codes into BFG2. The downside of using the program interface, however, is a loss of flexibility over models that conform to the component interface. Models written to conform to either the component interface or the program interface can be coupled together flexibly in any combination (which make scientific sense) using BFG2.

For more details on the component and program interfaces see Section 3. Currently, the BFG interfaces are most fully supported in FORTRAN90; however, equivalent API's are being developed for other languages, including Fortran77, C and Python. Compliant programs written in any of these languages can be coupled together. See Section 5 for progress on support for other languages.

2.3 Modularisation

The component API of BFG2 supports the idea of much finer grain code modularisation than is typically currently implemented in Earth System Modelling[5], although the trend is towards higher levels of modularisation in the future (for example, there are efforts to create standard routines for components such as radiation).

BFG2 is able to take two component-compliant models and couple them together so that, following the BFG code generation phase, when deployed they run 'in-sequence' (i.e. one after the other) within a single executable (i.e. running in a single operating system process), using argument passing to exchange coupling data. This deployment will execute with the same efficiency as equivalent hand-written code. With a minor change in the deployment metadata (see Section 2.4), the models can be deployed so as to execute concurrently, each model in its own process, using, for example, MPI or OASIS to exchange coupling data. With this flexible and efficient support for controlling how models are configured for execution, modellers are potentially able to modularise their code to a much finer level than is currently implemented. For example, the individual physics routines in an atmosphere model could (and probably should) be considered as (component) models in their own right.

Whilst the modularisation of code is considered to be a benefit (for code development, maintenance and, potentially, performance) it is not a requirement in BFG2. Users are able to choose the level of modularisation that is natural for their application and scientific goals.

When adopting BFG2, it is suggested that model developers input and output their data in its

[5]Codes are typically modularised to the level of atmosphere and ocean but no further, although ESMF is beginning to demonstrate the use of finer grain modularisation.

raw form, that is, in the format that the model itself stores the data. This is in contrast to the definition of standard interfaces, where models are required to input and output standard fields. Any mismatch between the formats of data exchanged by models is supported via transformations. BFG2 treats transformations in the same way as it treats science (component) models and this provides for the same flexibility in their deployment as discussed above. The potential performance advantage of this approach is discussed further in Section 2.4. Again, this approach is a recommendation in BFG2 not a requirement and users can add (hard-coded) internal transformations to their code and/or define standard interfaces if they so wish.

2.4 Metadata Descriptions

BFG2 generates 'wrapper' code from metadata descriptions (captured in XML) of the coupled model that have been specified by a user. BFG2 does not require any access to the source code of models themselves, although model source (or model binary) is subsequently required in order to compile and link.

The XML metadata descriptions are organised as three main categories, each dealing with a separate aspect of the coupled model, and each having its own XML document:

1. **Definition** of a model's coupling interface. This describes the input and output data that the model requires and provides from and to other models. This information only needs to change if the model interface changes. It is invariant over all couplings. This information is expected to 'live' with the code itself (for example, in the same directory). There is one definition metadata document for each model in a coupled model.
2. **Composition** of models into a coupled model. This describes how the models are connected together to exchange coupling data and how the coupled model is to be initialised (i.e. how the coupling exchanges are to be started). Once a composition is defined, the science to be computed by the coupled model is fully defined. This information is invariant over any deployment of the coupled model. There is a single composition document for a coupled model.
3. **Deployment** of models in a coupled model onto the underlying resources. This describes how the model codes are to be mapped onto the underlying hardware and software, including defining the specific (external) coupling framework to be used to implement the exchange of coupling data etc. Deployment metadata essentially describes how many processes will be required (at run-time) and how the models and transformations described in the composition will be mapped to the processes. Program code is then generated by BFG2 that conforms to the mapping specified. BFG2 may generate separate programs which invoke a set of models (calling each sequentially, 'in sequence') or BFG2 may generate an SPMD-style program that calls certain models depending on the run-time id of the process concerned. The process id will be the MPI rank in an MPI-based deployement.

   In addition, BFG2 requires the definition of a *schedule* which describes the control structure within which the models execute (at a relatively high, loop-based, level). The schedule provides a global view of the control loop structure of the coupled model and specifies an order in which models are to be executed *if* they are deployed such that they run in the same process. Concurrent execution of models can be implemented in an SPMD fashion, based on the schedule. Essentially, each SPMD program executes the whole schedule but, depending on the unique id of the program, only certain models are invoked during execution. BFG2 will add the logical conditions to ensure the correct models are invoked for each process generated.

   One example of the flexibility in deployment provided to the coupled model developer by BFG2 is that the developer can change from using one framework to another *without changing* the model code, the model descriptions, or the definition of the

science of the coupled model. Changing the coupling framework requires only a minor change in the deployment metadata, typically involving a single element[6], followed by a re-run of the BFG code generation phase and a rebuild of the coupled model, as necessary. There may be several deployment documents for a coupled model, each deploying the model onto difference resources.

The above separation of information is called DCD (Define, Compose, Deploy). For simplicity, BFG2 also uses a **coupled** XML document which contains references to each of the XML files associated with a particular deployment. Examples of coupled model metadata files are available online in the BFG2 portal (see Section 6 for details of the portal).

In order for BFG2 to generate the code to implement an exchange of coupling data correctly, it must be able to identify the metadata describing the data to be exchanged between two models (data required by one model and provided by another). That is, BFG2 must be able to relate information about coupling fields in the model Description metadata (and therefore to the actual data in the code), to statements about the fields made in the Composition metadata describing the required coupling.

In particular, BFG2 needs to be able to distinguish data provided (i.e. output) by a model and data required (i.e. input) by a model. The mechanism used to relate metadata to the code is as follows: BFG2 can determine which model provided, or requested data (whether via put/get calls or through the use of argument passing) as it knows which BFG model the request came from and it knows which models are currently 'active' – since BFG 'owns' the control code invoking models. Further, BFG2 can determine the specific data that is being input (or output) to (or from) a model from the data's position in the argument list - for an argument passing implementation - or from the 'tag' associated with the data – when put and get calls are being used. The tag and the position in the argument list are the link between the data in the model code and the description of the data in the metadata definition document.

In BFG2 the aim is to minimise the embedding of model metadata implicitly in code and instead to keep it as a separate description. The issue of maintaining consistency between code and metadata is discussed in Section 7 . This approach is consistent with the aim of separating infrastructure code from science code, providing the benefit of allowing the infrastructure to change independently of the science code and also providing flexibility in the choices of mapping to the underlying computational resources.

One example of BFG's flexibility is that if a coupling between two models requires a transformation on the data exchanged between them, the coupled model developer is free to choose where the transformation should be placed in the deployment, with potential performance implications. For example, if the models concerned are deployed in such a way as they run in separate processes, the transformation could be deployed so that it runs within the same process as either model (executing 'in-sequence'). Alternatively, the transformation might be deployed so that it runs in its own, separate, process. Further, if a model and a transformation use argument passing to exchange coupling data, and the deployment specifies they should share the same process, BFG2 is able to link the two codes together in exactly the same way, and with the same performance, as would have resulted if the model developer had hand-embedded the transformation into the science model directly. The deployment choice that gives the best performance will often depend on the underlying hardware and the flexibility in deployment provided by BFG2 can help in the porting of a coupled model to new machines.

[6]Other elements in the deployment may need to change depending on the target framework. Some frameworks have limitations on the mapping to executables/run-time processes. For example, OASIS3 requires one executable per model and ESMF requires one executable for all models in a coupled model.

2.5 BFG2 tools

The BFG2 code generation engine takes a coupled xml document (which contains references to all the metadata files describing a coupled model) as input and generates the required wrapper code and scripts for the coupled model. The way in which BFG2 code generation is implemented is discussed in Section 8. BFG2 can also be run online in the BFG2 portal, described in Section 6.

Having a description of a coupled model in XML also allows for other tools to be relatively easily written. BFG2 currently provides the following:

1. Stub generation. Model stubs can be generated from the model descriptions. Stubs are the coding of a models coupling interface without any (scientific) content. Actually, for testing purposes, this tool does addionally generate some code to provide values for output data so that this stub code can be compiled, linked and run with the BFG2 generated coupling code. This tool currently only works for Fortran codes.
2. Makefile generation. BFG2 does not keep any compiler specific information or any information about the structure of the codes (and how to make them). However, despite this limitation, most of the required Makefile can be generated.
3. Visualisation of the coupled model. The connectivity of a model can be translated into graphml (a well supported standard XML format for graphs[7]), and a tool is currently being developed to visualise the coupled model schedule defined in the metadata.
4. BFG1 to BFG2 translation. Models written to the old BFG1 structure can be used in BFG2 without any code changes. All that is required is for the previous BFG1 metadata description to be updated to the new BFG2 format. This tool does this automatically.

The combination of stub generation, Makefile generation and BFG2 wrapper code generation is used in some of the BFG2 unit tests. The combination of these tools allows the creation of model stub code, the creation of wrapper code, and the creation of a makefile to compile the codes. A hand written top level Makefile runs these generation tools, compiles the generated code using the generated Makefile and runs the resulting executable(s), each in its own process. This combination of tools allows rigorous development testing of BFG2 functionality, requiring only the definition of the appropriate metadata and a simple top level Makefile.

[7] http://graphml.graphdrawing.org

3. Supporting Aggregation and Composition

3.1 Current ESM Coupling Technology

The coupling technologies currently employed in Earth System Modelling (ESM) can be naturally split into two main categories[8]. In this document we term these two categories **aggregation** and **composition**.

In the first category the coupled model is an **aggregation** of pre-existing program-based model codes which (necessarily) run as separate concurrent executables (i.e. in separate run-time processes). In this case these model codes pre-exist and can (and typically will) also be run separately in their own right.

The main **advantage** of the aggregation approach is that it requires minimal intrusion into, or restructuring of, existing "legacy" codes and the underlying processes associated with running them. This benefit is significant in ESM as the major centres have very large "legacy" codes which typically require a significant investment in effort to modify. These centres will therefore prefer to take the path of least resistance, particularly as infrastructure changes do not immediately improve the science of the underlying codes, and science results are the main priority of modelling centres. As a result, the coupling system synonymous with the aggregation approach, OASIS3[9], has enjoyed widespread uptake, particularly amongst the European modelling centres.

The main **disadvantage** of the aggregation approach is the potential performance overhead when compared with hand-crafted (or composition-based, see below) solutions. The principal reason for this is that the mapping of the model codes in a coupled model to the underlying software and hardware resources on which the coupled model will execute is constrained, and this can lead to less efficient implementations. For example, where two model codes must run one after another (in a timestep), due to the requirements of the numerical algorithm implemented, the running of these on separate (sets of) processors - which is the natural way to schedule two (MPI-based) programs on a parallel computer - will result in a waste of resources. Further, in cases such as this, message passing is the most efficient way to pass data but this is much less efficient than passing data by reference, which is not possible in an aggregated solution consisting of several separate programs.

In the second category, the coupled model is a **composition** of component model codes which are not implemented as separate programs. Component model codes consist of a collection of program units - subroutines, procedures or methods, depending on the language in which they are written. These component codes cannot run in a stand-alone fashion without a main program code being provided – the main program code can be thought of as 'wrapper' code. The wrapper code is, therefore, to be considered part of the coupling software, and it is responsible for calling the underlying component codes. The calling of user code by external software is termed **inversion of control** and is one of the defining features of a **framework,** as opposed to a library, for example. Thus component codes that require composition need a **coupling framework**, whereas component codes that are aggregated require a **coupling library**.

The main **advantage** of the composition approach is flexibility in deployment, that is the way in which component models are organised into separate run-time processes. Flexibility in deployment allows for more efficient implementations since component models may be run sequentially ('in-sequence, one after the other(in the same process, concurrently, in separate processes, or in some combination of the two. Further, when some component

[8] http://pantar.cerfacs.fr/globc/publication/proceed/2011/Proceedings_of_the_workshop_final.pdf

[9] The successor to OASIS3 (OASIS4) is similar to OASIS3 in its approach but has been designed to be more scalable. However, a combination of inertia at the modelling centres, issues with the OASIS4 interpolation functions and the ability to use multiple instances of OASIS3 to improve scalability, has hindered its uptake.

models are run sequentially in the same process, data may be passed between them by reference, thereby avoiding the overheads of data copying and increased memory use incurred in an aggregated solution. Additionally, components may be composed in a hierarchy (with one component model calling other component models), which supports the natural component hierarchy found in ESM. The best known proponent of this approach in ESM is the Earth System Modelling Framework (ESMF) which is being adopted as a standard in the U.S.A.

The main **disadvantage** of the composition approach is that the component codes must be written as a collection of subroutines, procedures or methods. The required restructuring of existing legacy codes may be significant, particularly if a legacy code is already a hand-crafted composition of components which need to be separated. However, for the development of new models, a component-based approach is recommended.

3.2 BFG2 Proposed Solution

Historically, in the development of coupling systems, such as OASIS and ESMF, there has been an assumption that one must choose between the use of aggregation and composition. However, as mentioned in Section 2.1, there is agreement in the coupling community that supporting both approaches within a single coupling system would be beneficial. Within ISENES, the prototype BFG2 has now been extended to demonstrate the support of both aggregation and composition (previously BFG2 only supported composition).

3.2.1 Component API and composition

The following is a generic example of what a component model, written to confirm to the Component API discussed in Section 2.2, might look like. This example is written in the Fortran90-style interface. Support for other languages is discussed in Section 5. A model written using the Component API is said to be **component-compliant**. Note that the model consists of the declaration of a number of entry points, each defining a (sub)routine, and each entry point providing coupling data (to the framework, i.e. out of the model) and receive coupling data (from the framework, i.e. into the model) as both arguments and through the use of in-place put and get calls (as both argument passing and in-place calls are supported in BFG2 component compliant models).

```
 1 module atmos
 2   use bfg, only : put,get
 3   implicit none
 4   private
 5   public :: init,iteration,finalise
 6 contains
 7   subroutine init(arg1,arg2,...)
 8     call get(data1,tag1)
 9     call put(data2,tag2)
10   end subroutine init
11   subroutine iteration(arg1,arg2,...)
12     call get(data3,tag3)
13     call put(data4,tag4)
14   end subroutine iteration
15   subroutine finalise(arg1,arg2,...)
16     call get(data5,tag5)
17     call put(data6,tag6)
18   end subroutine finalise
19 end module atmos
```

The above code is not dissimilar to that which would be produced had the model been written to conform to the ESMF interface. Data can be provided to and from the coupling

framework via arguments (although ESMF requires that data also be registered with the framework and then ESMF passes references to the data via arguments). Also, the model can be split into multiple subroutines through the definition of entry points. Note, that, as with ESMF, an arbitrary number of entry points are allowed.

It should be relatively easy to imagine having an ESMF wrapper layer above this interface which includes all of the ESMF specific code. Section 9.1 contains an example of generating an ESMF-compliant model using BFG2.

One particular difference to the ESMF interface is that BFG2 also supports the passing of data to and from the framework via "in-place" `put` and `get` calls. In-place calls allow data that is scoped and computed locally, perhaps in a deeply nested subroutine called from a top-level entry point, to be input and output. Support for this can make it easier for developers to integrate their code into the framework since less code restructuring may be required – the data does not have to be passed through the argument lists of the routines in the calling hierarchy, for example.

Another motivation for the use of in-place calls is the need to support diagnostics. Diagnostics may be computed at some low level in the code and the code developer might want to output this information immediately without having to wait until the end of the routine, potentially reducing performance, or allocate memory for the data which would exist for the duration of the routine and therefore increase the amount of memory required by the model.

3.2.2 Program API and aggregation

Existing support in BFG2 for in-place calls is used in the case of aggregation. The code below is an example of a program written using the program API, again using the Fortran90-style interface. A model written in this style is said to be **program-compliant.**

```
 1 program atmos
 2   use bfg, only : put,get, bfg_init, bfg_eos, bfg_finalise
 3   implicit none
 4   call bfg_init()
 5   call get(data1,tag1)
 6   call put(data2,tag2)
 7   do i=1,nts
 8     call get(data3,tag3)
 9     call work(...)
10     call put(data4,tag4)
11     call bfg_eos()
12   end do
13   call get(data5,tag5)
14   call put(data6,tag6)
15   call bfg_finalise()
16 end program atmos
```

The above code is similar in style to that seen in OASIS3 and OASIS4, in that the code consists of a main program, has in-place calls to input and output data from and to the coupling system and has some additional calls to initialise and shutdown the coupling system.

One difference with OASIS3 and OASIS4 is that in BFG2 models to not have to timestamp the data input and output. To support this different approach an end-of-step signalling routine is provided by BFG2 (bfg_eos). This routine needs to be invoked to inform the framework that the next step is about to start (so the coupling system can arrange for the 'clock' to be incremented, for example).

Again it should be relatively easy to imagine implementing a separate OASIS3 or OASIS4 layer underneath the above interface. Section 9.1 provides an example of targetting OASIS3 from BFG2.

3.3 Summary

BFG2 now supports both the aggregation and composition of models. Models written in a component- or program-compliant manner can be coupled together in any combination that respects the requirements of the target coupling system (for example, a program-compliant model cannot be directly coupled into a coupled model targeting ESMF). Examples are discussed in Section 9.1. The extension of BFG2 to include aggregation demonstrates that a generative approach is a viable way to allow future coupling systems to support both aggregation and composition and this, in turn, will allow users to choose the most appropriate (coding) style and target coupling system for their needs.

4. Framework Interoperability

As previously described in Section 2, BFG2 promotes the isolation of science code from coupling framework code (including the mechanism used to communicate coupling data). BFG2 allows the user to specify which communication or coupling framework they would like to use (this choice is termed the *target*). BFG2 then generates the appropriate wrapper code required to implement the coupled model using the target and this code can then be compiled with the science code and subsequently run as a coupled model. Thus, the user is able to choose between different coupling frameworks with no change to the underlying science code. This can be considered to be a step towards framework interoperability.

What BFG2 currently does not support is the inclusion of a model that has been written natively in one of the supported communication or coupling targets (and by extension BFG2 also does not support a model written in an unsupported communication or coupling framework). To put it another way, to achieve framework interoperability for a set of models, all of these models must conform to the BFG2 API.

There are a number of reasonably mature coupling systems in use at the present time in Earth System Modelling (ESM) and this is expected to continue to be the case for the foreseeable future. In fact, as ESM integrates more closely with other domains, the number of coupling approaches may well increase – the hydrology community currently use OpenMI, for example. This is not unexpected, since different solutions have their own merits and it is unlikely that a one-size-fits-all solution will emerge due to the increasingly wide range of scenarios in which a particular piece of science code (representing a particular model) is utilised.

The current standard solution to this issue is for model developers to write and support a separate API for each framework with which they need their model to run. In practice, model developers typically only actively support one coupling approach (the one they are currently using) and therefore their models can only be coupled with models written to use the same framework. If one were able to couple models together that were written natively to conform to different frameworks, many more models could be coupled together. This would, in turn, reduce the burden on the model developer to support multiple API's.

As mentioned in Section 2, the BFG2 approach is to propose that model developers write code to conform to a scientific API and therefore separate their science code from any specific framework code. This approach would, at least, make it easier to manually create and support new framework API's. In fact, this is what some of the more modular systems already implement in an informal way, a good example being the models developed at GFDL[10].

BFG proposes to go a step further and automates the generation of the appropriate wrapper and coupling code, thus relieving the burden on model developers to do so. In IS-ENES we are in the process of modifying the BFG2 code generation system so that it is also able to wrap individual models, or collections of models, and **export** them for use directly in a selected supported framework. This is in contrast to the original use of BFG in which the creation of wrapper code for a complete coupled model is the aim.

There are two main benefits to this approach:

First, a model developer is now able to couple a BFG2-compliant model with models written natively to use other frameworks. However, there is a restriction in that the whole coupled model must use the same (native) framework. Further, the coupling of the exported model to the native model(s) has to be done manually within the framework. For example, with a BFG2-compliant atmosphere model and two ocean models, one written natively to conform

[10]In a private communication with Balaji.

to ESMF and the other to conform to OASIS3, then the BFG2-compliant model could be exported to ESMF and then manually coupled with the ESMF-compliant ocean model. The BFG2-compliant model could also be exported to OASIS3 and manually coupled with the OASIS3-compliant ocean model. In these couplings, the exported model is simply equivalent to a model natively using the target coupling system.

Secondly, a powerful feature of BFG2 is that it supports communication between models, written to be compliant to the component API, via argument passing and can couple models together using argument passing producing code that executes with the same efficiency as the equivalent, hand-written code.

Combining the use of argument passing with the facility to export models is powerful. For example, a collection of (BFG-compliant) models could be exported, wrapped as a single model, and then coupled to other, native, models using the relevant target coupling system. The component models in the exported model would be able to perform internal coupling communication efficiently using argument passing. This option would allow much finer grain modularisation than is currently used in coupling systems that support only aggregation, such as OASIS3. In effect, this allows the combination of model *composition* at a fine-grain level (with coupling data communicated using argument passing) with model *aggregation* at a more coarse-grain level.

The current status of BFG2 is that there is a working export option targeting OASIS4 models. This export allows either a single model to be exported or a collection of models to be exported with their internal communication being implemented by argument passing, as described above. There is also a partially working export option for ESMF which is currently limited to the export of a single model. In the future, the export option for ESMF will be completed and support added to target OASIS3.

Conversely, there is an aim to investigate the possibility of being able to **import** a model written to conform natively to external coupling API into BFG. There are already examples where this process has been implemented manually. For example, the DIVA model is written in Java and communicates using TDT[11]. A manual “adaptor” code, which conforms to the BFG API, has been written and this adaptor mediates communication between BFG and DIVA. That is, the adaptor 'talks' TDT at one side and 'talks' BFG at the other side. Support for the exporting and importing of models are steps towards a solution to the problem of framework interoperability.

[11]TDT was supported as a target for BFG1 but has not yet been implemented as a target for BFG2.

5. Language Support

Originally, BFG2 was developed supporting only the coupling of models written in Fortran. BFG2 offers two Fortran interfaces. There is a `SUBROUTINE` interface for use with models written to the Fortran77 standard, and there is a `MODULE` interface for use with models written conforming to the Fortran90 standard. The reason for initially limiting support to Fortran was that most models in Earth System Modelling (ESM) are written in Fortran. However, as ESM models begin to be used in other scenarios, for example as part of an Integrated Assessment system, other languages come into play as other domains frequently develop models in languages other than Fortran.

In IS-ENES, in addition to support for Fortran, we are adding to BFG2 support for models written in the C programming language and for models written in Python so that models written in these languages can be coupled together in any combination.

To implement this support in a flexible and extensible way we have added the concept of a *base language* in the BFG2 metadata. The base language is the language in which the 'wrapper' communications and control code, generated by BFG2, will be written. A base language can be specified for each BFG2 deployment unit (a deployment unit is a group of models that will be run together from within the same main program – i.e. within the same process at run-time). This allows more than one base language to be specified for a particular coupled model if the coupled model is implemented as more than one deployment unit.

To support multi-language coupling, the BFG2 code generation software has been extended to include a model interface generation phase. For each deployment unit the BFG2 generation software compares the (metadata-) specified base language and the specified language in which the model has been written, and it generates appropriate *adaptor* code to mediate exchanges between the two, if required.

The current status is that the base language must be specified as Fortran and models written in Fortran, C and Python can be coupled together by the generation of appropriate adaptor code (for the cases of models being written in C or Python - no adaptor code is required for Fortran models, as that is also the base language).

Clearly, if most, or all, models were written in Python, for example, it would make more sense for the base language also to be Python. Work in a separate EU project, called ERMITAGE (http://ermitage.cs.man.ac.uk/), is in progress to support Python as a base language. At this point in time, there is no plan to support C as the base language (since there is no user/developer domain requiring this).

There are clearly more languages being used to write model code than just Fortran, C and Python. For example, Java and R, are used to write model code in some domains, and BFG2 can be expected to extend its support to other languages in the future, as demand dictates. However, the fact that a particular language is not currently supported, does not mean that a model written in that language cannot be used with BFG2. Most languages have a defined interface to either Python or C, so users would be able to write their own adaptors to enable their models to be used as BFG models.

This approach is the same as that described above for the DIVA mode. It is possible for a user to write an adaptor to intercept communication to and from external models to make the user's model work as a BFG2 model. The DIVA model is written in Java and communicates using TDT. In another project, the authors developed a hand written Fortran adaptor model (called DIVINE) which communicates to the DIVA model using TDT and which is itself a BFG-compliant model. Thus, DIVA can be used with this adaptor as a BFG-compliant model.

6.BFG2 Online Access

A portal is being developed to allow online access to the BFG generation tools, input XML metadata files and (where appropriate) model source code. The portal can be accessed at http://bfg.cs.man.ac.uk. When the portal is more mature it will be linked-to from the ENES portal. Some advantages of providing a portal are that it allows easy access to the tools, with no need to download the BFG distribution and install any dependant libraries etc. and it provides access to pre-existing BFG2 examples for potential users and developers to browse.

The current version of the portal supports the following:

1. The upload of BFG coupling files (i.e. the XML files that describe a BFG coupled model),
2. pre-existing BFG coupling example files,
3. the validation of BFG coupling files,
4. the viewing of uploaded BFG coupling files,
5. the running of BFG to generate code from valid BFG coupling files,
6. the generation of model code stubs from BFG model description files,
7. the viewing of generated code,
8. the download of generated code,
9. separate sessions to allow for concurrent access to the tools by multiple users,
10. an examples repository which provides access to BFG coupling descriptions, BFG composition descriptions, BFG deployment descriptions and BFG model and transformation descriptions.

Work is ongoing with the portal and the following functionality is in development:

1. the editing of uploaded BFG files. This will allow users to change properties (such as target platform) and fix any errors in the XML, reported, for example, by the verifier tool, etc.,
2. the running of the BFG Makefile generator tool,
3. the running of the BFG1 to BFG2 translation tool,
4. graphical views of the BFG coupling descriptions (using existing BFG translation tools which convert BFG descriptions into visual representations).

Eventually, it is anticipated that the portal will enable users and developers, should they wish, to provide links to their BFG model code and transformation codes, where possible. The plan is to allow users to upload examples to the repository to promote community sharing.

In future, if there is demand from users, the BFG tools will also be made available as web services. This will provide a suite of web-based tools which could be used as part of a larger, more formal workflow. Finally, in the longer term, the aim is to use the CIM developed initially in the METAFOR project, to describe coupling configurations, but at the moment not all the required properties are present in the CIM, so further development of the CIM is required.

7. Code Parsing-Code and Metadata Consistency

One of the issues of having metadata describe model code and model couplings (as is championed in BFG) is that the metadata and the code that it describes can become inconsistent. One potential solution to this consistency problem would be to keep the metadata with the code itself. At least then, code developers would be more inclined to update the metadata when they updated their code. This solution is similar to requiring a developer to comment the code 'properly', and the problems of consistency between comments and code are well know. Thus, a tool which checked for consistency between metadata and code would be useful if not mandatory.

A key element in such a tool would be a robust code parser. A parser would be required either to support the extraction of metadata from code, or to compare the metadata with the code. A promising open source parser is the Open Fortran Parser (OFP) and in the recent EU-funded METAFOR project, to demonstrate the use of parser to help produce metadata from code, this parser was modified so that it output an XML representation of an input (Fortran) code. An XSLT translation phase was developed in order to create an outline METAFOR CIM document for the code based on the XML output by the parser.

In IS-ENES, the OFP solution has been extended and the OFP parser itself has been modified so that it correctly parses Fortran comments (previously it simply ignored them) and these are now also output as part of the XML output. This is a necessary step if metadata, perhaps embeded in Fortran code by developers as structured comments, is to be extracted.

Further, during the IS-ENES project, it has been agreed with the authors of OFP that the code developed at Manchester will be added to a branch of the OFP repository. As a result, the prototype Fortran-to-XML translator is now available as part of the OFP project.

8. Using Templates for BFG2 Code Generation

BFG1 and BFG2 were originally written using the well known XML translation language XSLT to generate code, scripts and (input data) files, as appropriate, from the BFG XML metadata files describing models and couplings. Two main issues emerged related to limitations of the XSLT-based approach:

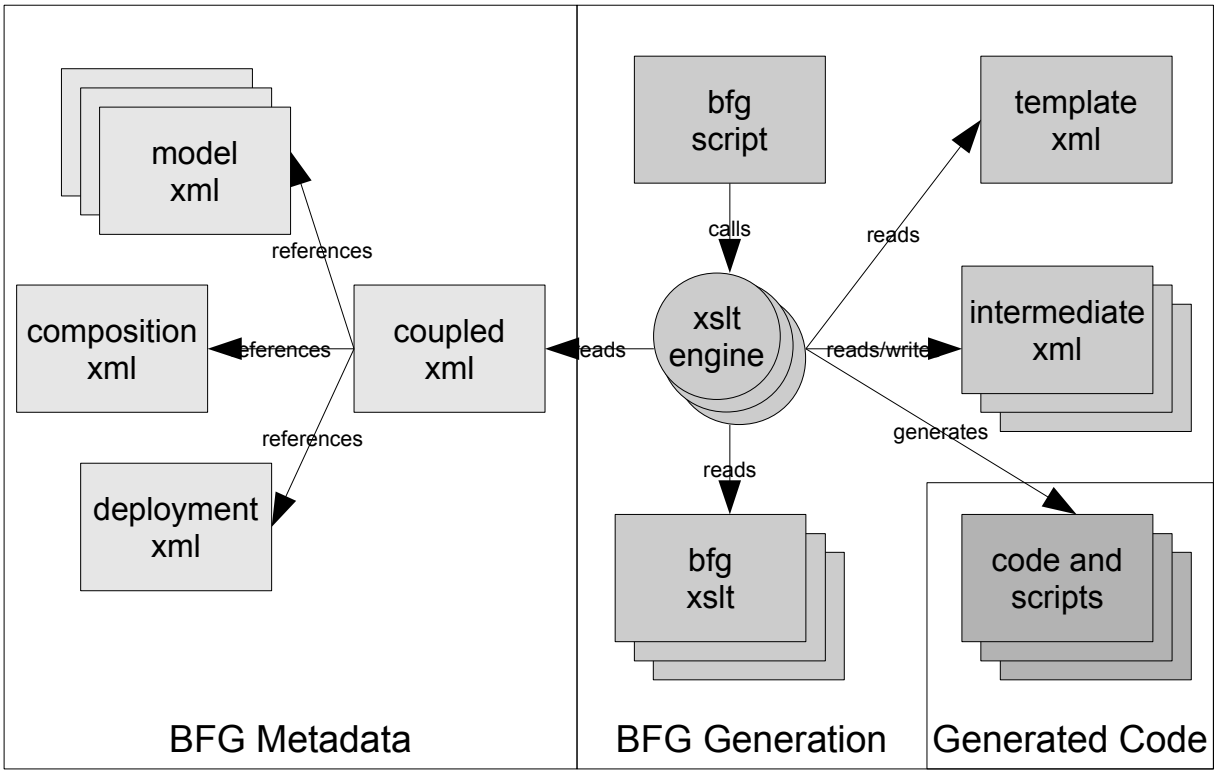
1. maintainability and extensibility of the resulting code generation engine,
2. performance of the engine as the complexity of the coupled model described increases.

These issues are discussed in the following two sections and the proposed new template-based solution designed to overcome these limitations is presented in the subsequent section.

8.1 Maintainability and Extensibility

XSLT is very good for describing simple transformations, however when more complex transformations are needed it can become complex, both to read and maintain. This is, in large part, due to its single assignment, functional implementation.

In order to simplify the XSLT code development, the code generation process in BFG2 was split into stages or phases[12]. A small number of XML templates were defined which were gradually filled out over a series of stages, thereby reducing the complexity of the overall process.



Whilst this approach is a good idea in theory, in practice most of the stages were relatively simple (for example adding the program name to an intermediate XML code) but one or two stages were very complex and it was not clear how to decompose these further sensibly.

A additional feature of the development approach taken was to keep all code description in

[12]BFG1 was simple enough to have a single phase

an implementation language-neutral XML syntax until a final code generation phase. The potential advantage of this approach is that it should be simple to change code generation from one target language to another. This would be achieved by simply changing the final code generation phase. The disadvantage is that a new, intermediate, language needed to be developed, supported and understood.

Again, this was a good idea in theory but the fact that most codes in the ESM domain are written in Fortran meant that the added complexity was not actually required[13]. Further, generated code in different languages potentially needs to be written in very different ways, so it is unclear whether a simple change in the code generation phase would be sufficient.

8.2 Performance

For simple couplings BFG code generation using XSLT is very fast. However, for complex couplings, particularly when argument passing is used to exchange coupling data, code generation times could take as long as 15 minutes. In early versions, generation times were even longer than this but the engine code was optimised to pre-compute as much as possible in a single step, and then re-use this information in subsequent steps. This re-design made the BFG XSLT translation code harder to understand and therefore more difficult to develop and maintain.

8.3 Solution

The proposed solution to the above issues is a template-based approach. In a template, one writes the generic code and adds markup where any specific code is required. For example, the template in Figure 1 uses the "%" symbol as a delimiter for the attribute `progname`. This template would be stored in a file, in this example a filename `programtemplate.txt`, is assumed.

```
program %progname%
end program %progname%
```

Figure 1:

The template engine then provides a simple way to replace the markup with text. The pseudocode in Figure 2 demonstrates the approach. Figure 2 contains two template commands which would be executed by the template engine. The first command tells the template engine which template is to be processed. The second command is an example that contains an attribute specifying the 'value' that is to be used when applying the `render` command to the template.

```
template=Template('programtemplate.txt')
template.render(progname='atmos')
```

Figure 2:

Finally, Figure 3 presents the output resulting from running the template engine with the template commands in Figure 2.

[13] Although, the requirement for framework interoperability, discussed in Section 5.may mean this decision may need to be reconsidered at some point.

```
program atmos
end program atmos
```

Figure 3:

An advantage of the template approach is that it is possible to view and edit the static part of the template code simply, with no change to the template engine. There is also no need for a language-neutral format; instead different templates are written for different languages. There are, therefore, potentially a large number of templates but each template is simpler to understand and manage than the previous XSLT-based, phased approach.

There is also no longer any need for XSLT. Instead, all of the code is written in Python. Python was chosen because it has all the functionality required and is easy to program. The role of the Python code is to read in the BFG2 XML documents and translate it into appropriate template commands and attributes and then apply the commands to the pertinent templates containing generic code.

To improve performance, the translation from BFG2 XML to template commands and attributes is performed on-demand so that translation is only performed where required. Further, the results of any translation are stored so that any future need for a particular set of commands and attributes does not need any further translation. This is one of the reasons why the template implementation is expected to be much more efficient than the previous XSLT-based approach.

In the first instance, the templating engine that has emerged was designed to directly support the requirements of the BFG metadata translation problem as they emerged during development. However, there are a number of well supported Python-based template languages already in use which may have the functionality required. Some examples are: StringTemplate[14], Jinja2[15] and Genshi[16]. In future, development may migrate to one of these languages.

8.4 Status

The XSLT version of the BFG code generation is split into 3 main sections

1.control code generation: this section creates the required main program(s) code, calls the scientific and transformation models respecting the specified schedule, performs any required initialisation and shutdown functions, declares and initialises all argument passing data and ensures that any coupling that involves argument passing data is performed correctly. This code is written in a target neutral manner using a generic interface so does not need to change if the underlying communication target changes (for example changing from OASIS3 to OASIS4).

2.inplace code generation: this section creates the library that satisfies the in-place (put get) communication requirements. It determines what data is being provided and routes that data to the appropriate place. For performance reasons it does call the underlying communication mechanism (for example, MPI send, or prism_put) directly so much be regenerated if the communication target changes.

3.target-specific code generation: this section creates the target specific communication interface. It implements the generic interface used by the generated control code. Different implementations are created for different targets. For example, for MPI it

[14]http://www.stringtemplate.org

[15]http://jinja.pocoo.org

[16]http://genshi.edgewall.org

implements the framework_init() function by calling MPI_INIT (amongst other things) and for OASIS4 it implements the framework_init() function by calling prism_init() (amongst other things).

In the template version of the BFG code generation the control code generation and in-place code generation sections of BFG have been replaced with template versions. The authors are currently working on the control code generation version to add in full support for argument passing. The target specific code generation has been left using the old XSLT approach for the moment.

One of the motivations for moving from XSLT to templates was for easier extensibility of the framework. At the same time as re-writing BFG to use templates, support for models written in C and Python are also being added. Please see Section 5 for details. Further, the template version supports the new program compliance option, see Section 3 for details.

Another motivation for moving from XSLT to templates was to get high enough performance to make BFG available as an online tool. This work is ongoing, please see Section 6 for details.

When the template code implementation has been completed the code will be made open source and added as a SourceForge or GoogleCode project.

9. Examples

9.1 OASIS and ESMF as BFG2 targets

In this example we present the coupling of four toy models which are designed to represent some of the basic constituents of an Earth System Model. Both the individual models and a composition of the models are described. The composition includes a number of transformations performed on coupling data exchanged between models and these are used to illustrate how transformations can be handled in BFG2. A number of examples of deployment of the coupled model are given which demonstrate the power of BFG2 – simple changes are made to the deployment metadata describing the model and the BFG2 code-generation produces the 'wrapper' code required to implement the model. The example is used to show how alternative coupling technologies, such as OASIS or ESMF, can be selected to implement the exchange of coupling data between models deployed to run in separate processes. The example also shows how models deployed in a single run-time process can communicate using efficient argument passing. Further, the ability of user to map models to run-time processes in a flexible way, and thus exploit concurrency in the deployment of a coupled model, is demonstrated. Finally, the ability to 'export' a model (or set of component-compliant models) from a composition so that it can be used as a stand-alone model, using a specific coupling technology, is illustrated.

The functionality described in the Section is in the process of being added to BFG2. The summary section (Section 9.1.5) describes what is currently supported.

This code for the toy models and transformation used in the example and the BFG2 metadata associated with the example is is expected to be available by the end of April 2012 on the BFG web portal described in Section 6.

The toy models used to illustrate BFG2 represent an ocean model, a dynamics model, a convection model and a radiation model. The dynamics, convection and radiation models would typically currently all be found embedded within an atmosphere model. They are defined separately in this example to illustrate the potential benefits of a finer grain approach to modularisation in ESMs. The codes are all written in Fortran90. The dynamics, convection and radiation models are component-compliant and, hence, are written to pass all their coupling data by argument passing. The ocean model is program-compliant and (necessarily) is, therefore, written to pass its coupling data using only in-place put and get calls.

9.1.1 Model metadata

The basic details of the dynamics model are given in the table below. It has a timestep of one hour and consists of two subroutines (or entrypoints): an initialisation subroutine and a timestepping subroutine. All input and output data associated with these subroutines is available through the argument lists because the model is component-compliant. The data id in the final column of the table indicates the position of the data in the argument list (this id is used by BFG2 to link metadata to data uniquely, as described in Section 2.4). Similar descriptions exist for the other models.

Model name	Language	Timestep	Entrypoint name	Data form	Data direction	Data id
dynamics	f90		init	argpass	out	1

Module				argpass	out	2
				argpass	out	3
				argpass	out	4
		1 Hour	dynamics	argpass	inout	1
				argpass	inout	2
				argpass	inout	3
				argpass	inout	4

The following table presents a summary of the main metadata information for each of the 4 models. One point to note is that in this coupled model, the radiation model, as it typical in ESMs, runs less frequently - with a longer timestep - than the other components.

Model name	Entry points	compliance	timestep
dynamicsModule	2	component	1 hour
convectionModule	1	component	1 hour
radiationModule	1	component	4 hours
oceanModule	1	program	2 hours

9.1.2 Composition metadata

Figure 1, summarises how the models are composed into an example coupled model. This diagram is generated from the BFG composition metadata by a BFG utility routine (bfg2graphmlgen.py). The utility routines currently available with BFG2 are described in Section 2.5. This is an example of the benefits of having metadata descriptions of models and compositions; several tools exist which can manipulate and display the metadata in various useful ways. Other examples are given below.

In Figure 1, the scientific models are represented by the (light blue) rectangular boxes and the (orange) circles represent transformation models (or transformers) that have been added to manage the exchange of coupling data between models running with different timesteps. The number on each line connecting models (and transformers) in the figure represent the number of data fields being passed between them.

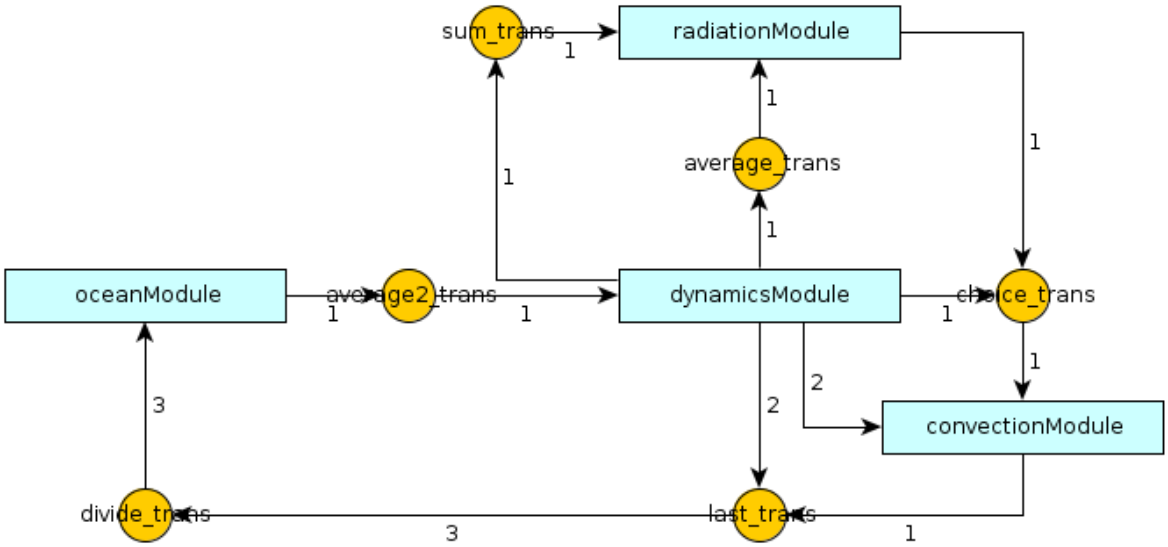


Figure 1: Summary of the Composition

Different types of transformation have been purposely included in this example as an illustration of the type of processing of coupling data frequently required in coupled models. For example, the dynamics model passes data to the radiation module. However, the radiation module runs at a slower rate than the dynamics module (the radiation model has a longer timestep), so, in this example, the data from the dynamics model is summed by an appropriate transformation and the sum is passed to the radiation model when it does run.

A more complex transformation example is shown in the case where a model, running on a particular timestep, can receive data from one of two models, depending on which one is running on the same timestep. For example, the radiation model passes data to the convection model on the timesteps when it (the radiation model) is running but the dynamics model passes data to the convection model when the radiation model is not running. A (priority) choice transformation is used to implement this behaviour. The transformations used in this example were written specifically for this coupled model. However, a library of standard transformations will be added to the BFG2 repository so that users are able to make use of them in compositions.

After the composition metadata is completed, the science to be computed by the coupled model is defined. The next section describes how the coupled model may be deployed onto a set of computational resources (software and hardware) in a flexible manner.

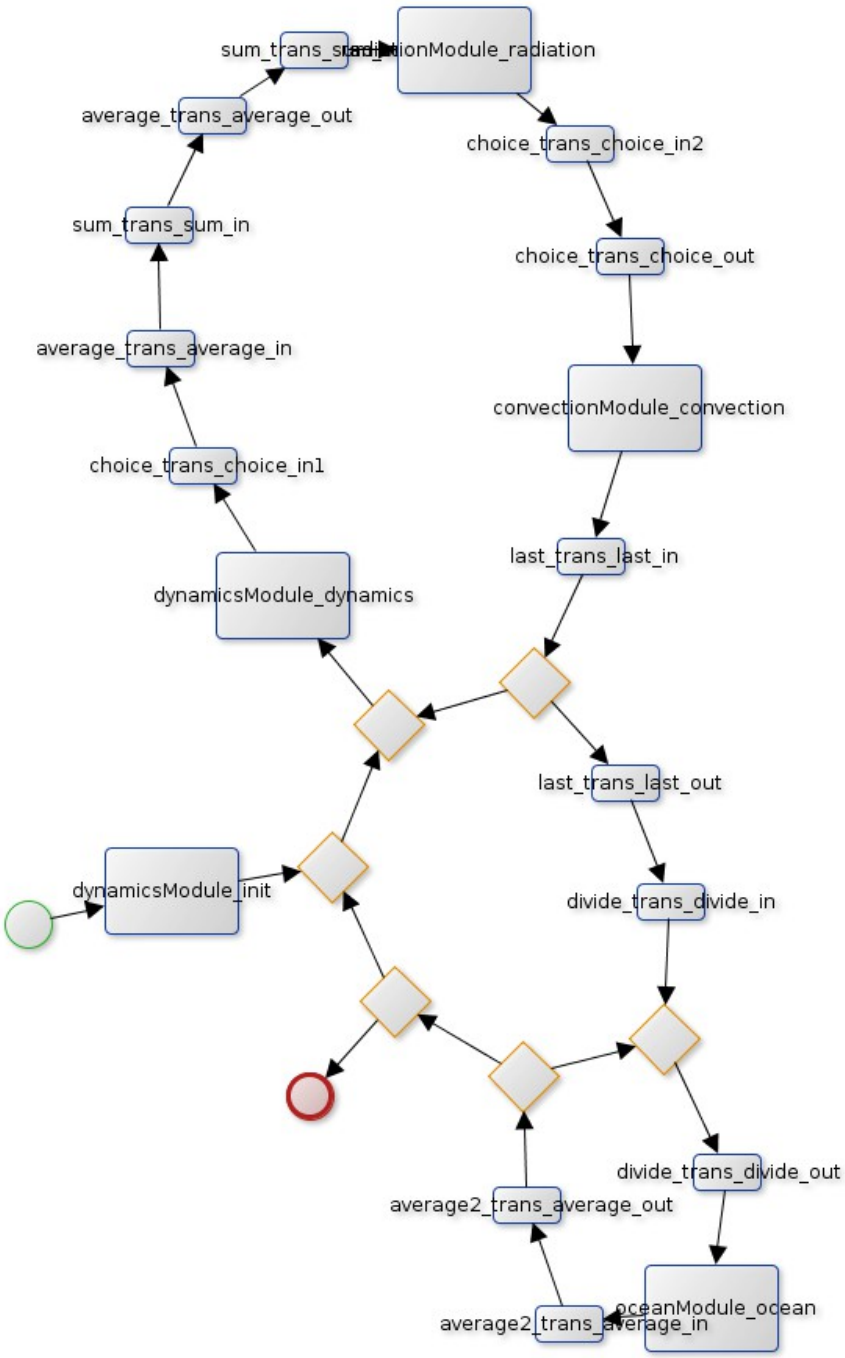


Figure 2: BPMN representation of Schedule

9.1.3 Deployment metadata

An introduction to the BFG2 deployment metadata is presented in Section 2.4. This metadata includes the definition of which specific coupling technology is to be used to exchange data between models which are deployed to run in separate (operating system-level) processes. Models may execute in different processes either because they are deployed in separate programs or because they are deployed in an SPMD-style in a single program. Program-compliant models are already separate programs, and this reduces their flexibility in the

deployment phase. However, for component-compliant models, there is flexibility.

Deployment metadata also specifies how many processes BFG2 is to deploy models to, and which component-compliant models will be deployed within each process. Models deployed within the same process can use argument passing to exchange coupling data efficiently through shared memory. Models deployed in separate processes must use the selected coupling technology (for example, OASIS or ESMF) to exchange coupling data. BFG2 takes appropriate action, based on the metadata, to generate wrapper code that ensures the coupling between models deployed as described in the deployment metadata is implemented correctly.

As described in Section 2.4, the deployment metadata also contains a schedule for the coupled model. Figure 2 presents the coupled model schedule for this example using the Business Process Modelling Notation (BPMN). This diagram is generated from the BFG deployment metadata by another BFG utility routine (bfg2bpmngen.py) – see Section 2.5 for a description of other utilities available.

In Figure 2, the entrypoints of the scientific models are represented by the larger (blue-edged) rectangular boxes and transformation entry points by the smaller (blue-edged) boxes. The names in the boxes are the concatenation of the model name and entrypoint name for the particular entrypoint. The light (green-edged) circle indicates the start point for model execution and the bold (red-edged) circle indicates the termination point. The control flow specified for the model is shown by the directional arrows and the diamond shaped (yellow) boxes represent the start and end of loops in the schedule.

In this schedule the first action is to call the dynamics model initialisation routine. The main timestepping loop is then entered with a timestep of one day. An internal timestepping loop (consisting of 24 steps, each of one hour) calls the dynamics, radiation and convection entrypoints and each of the associated transformations (at the appropriate rate). A separate internal timestepping loop (this one consisting of 12 steps, each of two hours) calls the ocean entrypoint and the associated transformations. There are two transformations that span the two internal loops. These support the coupling between the ocean model and the other models and this coupling happens once each day.

9.1.4 Examples of flexibility in deployment

In this section, examples of flexibility in the choice of target coupling infrastructure and in the mapping of (component-compliant) models to run-time processes is discussed. Finally, the ability to export a composite model from BFG2 so that it can be used with other models built to use a specific coupling infrastructure is discussed. For example, the radiation, convection and dynamics model (along with the appropriate transformations) can be deployed as a composite (atmosphere) model using any target coupling infrastructure – OASIS or ESMF, for example. The resulting composite model is equivalent to an atmosphere model hand-coded as an OASIS or ESMF model and can be coupled to an Ocean model explicitly developed to use one of those coupling systems.

Flexibility in choice of coupling technology and in the number of processes to use

By changing the target coupling infrastructure specified in the deployment metadata, and by changing the mapping of models to processes, which is also specified in the deployment metadata, the coupled model code can be deployed in a number of ways. The flexibility in deployment is obviously limited by the compliance levels of the models concerned. Program-compliant models are less flexible than component-compliant models.

The ocean model is program-compliant and, therefore, already consists of a main program and must be run as a separate program in its own process. Note that in this discussion we are ignoring the fact that any of the models may exploit 'internal' parallelism and so use several processes, or threads, in their execution, depending on how the internal parallelism is implemented – using MPI or OpenMP, for example. It is assumed in this discussion,

without loss of generality, that the models are sequential models.

The dynamics, radiation and convection models, and all of the transformations, are component-compliant. This provides for a range of possible deployments for these models. For example, by defining appropriate metadata these models may be run in-sequence, one after the other, from within the same process. Alternatively, each model may be made to run in its own process . Any mapping of component-compliant models and transformations to processes can be defined, the choice often being made on performance reasons – it may be better to deploy certain transformations so that they are applied in the same process in which a model runs; for example, an interpolation routine might best be run on a fine-grain grid so that the interpolated data on a coarser grid is communicated to another model.

BFG2 will generate code and data declarations such that models that run in the same process will pass coupling data efficiently using argument passing and if they are run in different processes they will pass data using the target coupling technology specified in the metadata.

As a more concrete example, if OASIS3 were chosen as the target infrastructure and two main programs (i.e. two main programs each executing on a single process) were requested in the deployment metadata, one for the ocean model (which must run as its own program) and one for the remaining models and transformations, then BFG would generate the required wrapper code for the two main programs and the communications code. In this case, argument passing would be used for all communication between the dynamics, radiation and convection models (and the transformations involved) and OASIS3 would be used for all communication between any of these models (and the relevant transformations) and the Ocean model.

If the coupled model developer changes the target coupling technology in the deployment metadata, say to OASIS4 or simply to MPI, then BFG2 would generate code to communicate with the Ocean model using that coupling technology. The change of coupling technology requires no change to any of the model and transformation code, nor to any of the metadata, other than the simple change to the deployment metadata.

At the moment it is not possible to generate a coupled model using ESMF for this example as we have a program compliant model which has to run in its own program, and ESMF does not support multiple executables.

BFG2 deployment terminology

The mapping of models to programs and processes is defined in the deployment metadata in terms of *deployment units* (DU) and *sequence units* (SU). Essentially, models placed in the same sequence unit will run in the same process at run-time. Models in different sequence units will run in different processes. A deployment unit basically maps to a program, so if a single sequence unit is associated with a deployment unit, there will be a single main program generated by BFG2. This will result in a single process at run-time and this process will execute the models. Models in the sequence unit will use argument passing to communicate with each other and they will run in the order specified in the schedule specified in the deployment metadata.

Alternatively, two deployment units could be specified in the metadata and the models in the original sequence unit split between two new sequence units, with one sequence unit associated with each deployment unit. In this case, the models within a sequence unit would communicate using argument passing and all communication between models in the different sequence units would be through the specified coupling technology. Models in each sequence unit would again run in the order specified in the schedule. BFG2 would generate two main codes and the appropriate communication code.

As a final example, the two sequence units in the last example could be associated with a single deployment unit. In this case, BFG2 would generate a single, SPMD-style main code

(similar to an MPI SPMD program). Two processes would be required at run-time, one process invoking the models in one sequence unit and the other process invoking the models of the second sequence unit. Again, models within a sequence unit would communicate using argument passing and models in different sequence units would use the specified coupling technology.

The mapping of models and transformations to sequence units and the association of sequence units to deployment units is a very flexible and powerful technology for controlling the deployment of coupled models onto software resources (process mappings and coupling technology) and hardware resources (processors). Different deployments of the toy coupled model are shown in the deployment metadata available on the BFG2 portal for this example.

BFG can be run using the command line runbfg2.py <coupled_model.xml>. Given the flexibility in deployment described above, a model developer can very simply change from using OASIS4 or OASIS3 or even to MPI, as the selected coupling technology. This flexibility would allow a developer to develop the coupled model on local resources before using a full-blown coupling system, without having to perform any re-coding of their scientific models and without having to make changes to model description metadata or to the composition metadata.

Exporting (composite) models ready to use a specific coupling technology

Program-compliant models or combinations of the component-compliant models in this example can also be exported as OASIS3, OASIS4 models or, in the case of individual component-compliant models, as ESMF models. For example the dynamics, radiation and convection models could be exported as a single, composite OASIS3 model, The exported model would be equivalent to an OASIS3 atmosphere model. This is achieved by running the bfg2export command which takes as arguments the target coupling technology and a specification of which models in the composition are to be exported as the composite. The result is a single program (containing the composite model) which looks and behaves just like a hand-crafted model using the same coupling technology.

This ability to create and export composite models allows much finer modularisation of codes than is required for, for example, an OASIS3 coupling. BFG2 automatically combines the fine-grain model in the composite in an efficient way, using argument passing to exchange coupling data between the models in the composite, and allows the resulting composite model to be run as a single OASIS3 model.

Currently, it is only possible to export individual component-compliant models as ESMF models. This is not too much of a restriction though, as ESMF is designed to couple together more modular codes in a similar manner to BFG (but using data references rather than direct argument passing).

9.1.5 Summary

In this example, a set of toy models have been described which can be used to explore the flexibility in coupled model composition and deployment. The toy models illustrate the use of both component-compliant and program-compliant models. Flexibility in deployment of models to run-time processes and in the choice of a specific coupling technology has been discussed and the ability to export models in a BFG2 composition as a composite model, ready to be coupled to other framework-specific models, has been explored. The source code and example metadata for the toy models, compositions and deployments discussed are available from the BFG2 portal described in Section 6.

The current status of BFG2's support for OASIS and ESMF is the following:

OASIS4: there is a working implementation which can be used to couple models which are component compliant and/or program compliant. The OASIS4 implementation also supports the export option allowing either a single model to be exported or a collection of models to be exported with their internal communication being implemented by argument passing.

OASIS3: there is a mostly working implementation of coupling which can be used to couple models which are component compliant and/or program compliant. The export option is also mostly working and this supports the same functionality as OASIS4. A few bugs need to be ironed out and a namcouple generation routines needs to be added.

ESMF: there is an early prototype ESMF option. This generates working coupling and export code but does not generate any required communications code at this stage.

9.2 Experiences with JULES, a land surface model

Introduction

Component compliance (supported by BFG and mandated by ESMF) gives more flexibility in deployment than program compliance (supported by BFG and mandated by OASIS3 and OASIS4) and offers the potential for higher performance through the use of data sharing rather than message passing in certain configurations, for example. It has been suggested that for a new model it makes sense to use component compliance. However the re-engineering cost of making existing models component-compliant can be considered to be prohibitive. The work in this section looks at the costs of making an existing model component-compliant in order to give an indication of the required re-engineering effort and the issues that must be faced.

This section outlines the issues faced when extracting two component-compliant versions of a Land Surface model from a larger code base. The first was extracted from the Met Office Unified Model (UM) at version 7.1 and the second from a stand-alone Land Surface model code called JULES at version 1.

Both models share the same heritage and have a similar structure. The JULES model itself was extracted from the UM at an earlier version and both versions have since been separately developed. JULES is primarily used and developed by the U.K.'s research community and is supported by the U.K.'s Natural Environment Research Council (NERC). The Land Surface model in the UM (the ancestor of JULES) has continued to be developed internally by the Met Office.

The underlying motivations for this work were:

1. To help create a single Land Surface model that could be used in the UM, as a standalone model and coupled to other models where appropriate.
2. To put inplace a structure that would help enable further development of the Land-Surface model; in particular, to allow the addition of the existing ED, ECOSSE and SPITFIRE models.

Two additional requirements that were specified by the JULES developers were: first, that the replacement JULES code should be as efficient as the existing UM Land Surface model when configured with the UM, i.e. modularisation should produce no performance overhead, and, second, that FCM (the Met Office's Configuration Management tool) would be used to share source, as this is what is currently used in the UM.

Determining the initial JULES interface

The first step was to determine the appropriate separation between the Land Surface science code and the control layer in the standalone JULES code and in the Land Surface code in the UM.

Figure 3 presents the analysis of the JULES code. The top level structure of the code was examined and the routines that were called were compared with those that were already in the UM code. The logic behind this approach was the knowledge that the code bases were similar at a high level. Figure 4 presents a similar analysis of the UM code. Note, the whole UM call structure is not presented in this figure, merely the appropriate subroutines.

```
JULES
  INIT
    ALLOCATE_ARRAYS
    INIT_OUTPUT
    INIT_VARS_TMP
    INIT_PARMS
      TILEPTS             <== UM ROUTINE (VEGETATION)
      SPARM               <== UM ROUTINE (VEGETATION)
      FREEZE_SOIL         <== UM ROUTINE (INIT)
      CALC_BASEFLOW       <== UM ROUTINE (SOIL) Only called by HYDRO in UM
      CALC_FSAT           <== UM ROUTINE (SOIL) Only called by HYDRO in UM
  CONTROL
    ZENITH                <== NOT A UM ROUTINE
    FTSA                  <== UM ROUTINE (RADIATION)
    TILE_ALBEDO           <== UM ROUTINE (RADIATION)
    SF_EXPL               <== UM ROUTINE (BOUNDARY_LAYER)
    SF_IMPL         <== UM ROUTINE (BOUNDARY_LAYER)
    HYDROL          <== UM ROUTINE (SOIL)
    VEG2                  <== UM ROUTINE (VEGETATION)
    VEG1                  <== UM ROUTINE (VEGETATION)
  OUTPUT
  DEALLOCATE_ARRAYS
```

Figure 3: JULES Scientific interface

```
initial
  initveg
    INIT_MIN        <== NOT A JULES ROUTINE
    TILEPTS             <== JULES ROUTINE
    SPARM               <== JULES ROUTINE
    INIT_ACC            <== NOT A JULES ROUTINE

Atmos_Physics1
  NI_rad_ctl
    tilepts             <== JULES ROUTINE
    ftsa                <== JULES ROUTINE
    tile_albedo         <== JULES ROUTINE

Atmos_Physics2
  NI_bl_ctl
    BL_INTCT
      BDY_LAYR
        SF_EXPL         <== JULES ROUTINE
  NI_imp_ctl
    IMPS_INTCT
      IMP_SOLVER
        SF_IMPL         <== JULES ROUTINE
  HYD_INTCTL
    HYDROL          <== JULES ROUTINE
  VEG_CTL
    VEG_IC
      VEG               <== JULES ROUTINE (called VEG1 or VEG2)

u_model
  UP_ANCIL
    UPDATE_VEG
      TILEPTS           <== JULES ROUTINE
      SPARM             <== JULES ROUTINE
```

Figure 4: UM Land Surface Model Scientific Interface

From the analysis summarised in the two illustrations the following subroutines routines were

identified as making up the JULES interface. This interface is termed the “science interface” as it provides an interface to the JULES code that implements the science. This interface was subsequently agreed with the JULES developers.

TILEPTS

SPARM

FTSA

TILE_ALBEDO

SF_EXPL

SF_IMPL

HYDROL

VEG1

VEG2

Storing Building and Running

To support the development of the JULES model a repository was set up on NERC's PUMA machine. PUMA also hosts the external mirror of the UM repository and is the gateway to running the UM on the U.K. Research Council's supercomputers. This repository is based on SVN and includes support for TRAC and a WIKI.

The JULES code was separated into two directories (src and wrapper), the first containing the science code and the second containing the control code to maintain a separation between the two and to allow the science code to be used with different wrappers.

The code was subsequently modified so that it could be used with FCM, the UM's version control and build system. The only code changes that were required were to do with the requirement to have one subroutine per file, and needing to add FCM dependence comments to codes that call other routines in the traditional Fortran77 style (i.e. not using modules).

At this point the JULES code could simply be checked-out and built on local resources using FCM from the central repository. The final structure of the repository is given below:

build/

docs/

examples/

src/

wrapper/

The UM land surface model was also extracted from the UM and placed as a different project in the same JULES repository, with a similar structure. Header files that were used only by the land surface model were also extracted. At the same time a branch of the UM was created which had the land surface code removed. An example UM job was created which used this branch of the UM and the land surface code from the repository. The only difference between a standard UM job and a job using the separate UM land surface model was the addition of two additional FCM lines in the job specification.

Modularisation

From now on the term JULES is used to mean either the JULES model or the UM Land Surface model since the modularisation work typically applies to both.

a) Modules

The subroutines in the scientific interface were placed in a single Fortran module named JulesModel for both the models discussed in this work (even though the code extracted from the UM is not strictly JULES). This approach was taken as modules provide a simple way of controlling access to subroutines: they provide some compile-time checks on arguments and, most importantly they support the encapsulation of data. Table 1 provides an overview of the way in which JulesModel was incorporated back into the UM.

UM file name	UM subroutine name	JULES UM interface
initveg1.f	INITVEG	USE JULESMODEL, ONLY :: TILEPTS,SPARM
rad_ctl2.f	NI_RAD_CTL	USE JULESMODEL, ONLY :: TILEPTS,FTSA,TILE_ALBEDO
bdylyr8a.f	BDY_LAYR	USE JULESMODEL, ONLY :: SF_EXPL
impslv8a.f	IMP_SOLVER	USE JULESMODEL, ONLY :: SF_IMPL
hyd_ic7a.f	HYD_INTCTL	USE JULESMODEL, ONLY :: HYDROL
veg_ctl1.f	VEG_CTL	USE JULESMODEL, ONLY :: VEG
update_veg.f	UPDATE_VEG	USE JULESMODEL, ONLY :: TILEPTS,SPARM

Table 1: Including the Land Surface Model back into the UM

When introducing the JULESMODEL module, one problem that had to be addressed was that the declaration of variables was not always consistent between the UM calling routine and the UM Land Surface model. For historical reasons, some integers were actually declared in the UM as reals. Further, the UM keeps all of its data in a large one dimensional array called D1. D1 was used to pass the space for multi-dimensional arrays in the UM Land Surface model, and the module compile time checks did not allow this. Therefore, the arrays were changed so that they were declared appropriately in the parent UM routine and passed into the parent UM routine in the old style.

There is another issue when using modules. Module names must be unique within a single main program. Therefore, one needs to avoid the possibility of name clashes with modules from other libraries. As a result, it has been proposed that all modules internal to JULES should add "_JULESMODEL" to the end of their name (or something similar) to effect a private namespace, although this has not yet been implemented.

b) No Communication via COMMON or MODULE's

For a model to be component-compliant it should not expect data input from other models, or provide data output to other models, using Fortran COMMON or Fortran MODULEs. The reason for this is that the use of COMMON or MODULEs for data sharing between models limits their composition to a single executable, necessarily with both models running in-sequence. It also potentially ties one model to another, breaking modularity.

To achieve component-compliance, any data that is input to JULES or output from JULES should be passed by argument (or through in-place put/get calls, if required). Ensuring that this is the case is not trivial as the JULES code is rather large. As a test, a dummy main program was created which called each of the subroutines, but which only declared data that was passed by argument. Thus any COMMON or MODULEs showed up as being unresolved. All cases found in this way were modified so that all data was passed by argument.

Note, it would be possible to support data sharing via MODULE's and/or COMMON by directly supporting it in the component-compliance API but this was considered to be one step too far for a coupling system, so coupling data support was limited to arguments and in place puts and gets.

One special situation where data can safely be passed via MODULEs is for constants, and it is expected that an additional module called JulesModelConstants will be added, at some point, which allows external access to the internal constants used in JULES.

c) Internalising JULES Configuration

The JULES models are really composite models. There were external switches in the UM and JULES wrapper code which selected certain science options, as well as allowing the selection of the rates at which certain parts of model would run. Further, in the UM, different science options were chosen at compile-time using pre-processing. For example, different types of vegetation could be chosen: the triffid scheme could be switched on or off and the rate at which the triffid scheme was called could be controlled.

It was decided to take an internal, run-time approach in the new, modularised code. All of the switches and rates were removed from the UM and the JULES wrapper and kept in the JULES code itself. This has the effect of reducing the knowledge that the UM has about JULES internals, thereby increasing modularity. Choices were also made at run-time rather than compile time. The advantages of this approach is that the whole code is compiled each time and this reduces the chances of bugs in the code, and the code does not need to be recompiled when any configurations are changed. The disadvantage of this approach is that the executable will be larger.

To support internal run time switches and rates, a new JULES initialisation routine was written which read the values of these switches from a JULES input file. The two types of vegetation routine in the interface were also rationalised to one routine and, internally to JULES, the appropriate one was chosen. In addition, a set of run-time checks were added to ensure that valid sets of switches were selected (this was missing in the original code).

JULES uses tiles to represent the proportions of vegetation types at each grid point. JULES supports different numbers of vegetation types and it is the UM that sets and manages (the arrays for) this. The setting of vegetation types was moved to the JULES initialisation routine where it was read from a file. This value was then passed back out to the control code (which may be the UM) which is then able to allocate the required data appropriately. Eventually, it is hoped that the UM and any other control code for the modularised JULES will become fully ignorant of the internal tiling decisions made by JULES.

d) Modifying the Argument Lists

The JULES internal subroutines each have a large number of arguments. These can be typically split into constants, container data, stash (D1) data and coupling data. The only data that really needs to be passed by argument is the coupling data. Constants can be read in by the initialisation routine, either by argument or from a file. Container data is data that is declared outside of JULES but is only used internally by JULES. This data should be managed internally. Stash data is data that is going to the UM diagnostics system (which is called Stash). This can be passed by argument but is perhaps better passed using in-place 'put' calls.

Modifying all the argument lists appropriately is quite a big task and work has only just started on this. As things stand, most of the constant data has either been made internal to JULES or is passed in by argument in the JULES initialisation routine, in addition some of the container data has been internalised.

e) Initialisation and Restart

The timestepping data that is internalised in JULES will need to be able to be written out to file (i.e. a JULES 'dump' needs to be supported) and JULES will need to be able to perform a restart from a named JULES dump file. Initialisation and restart routines have been written using NETCDF as the underlying format to read in any JULES dump data. Similarly a JULES output dump routine has been written to support the writing of JULES dumps in NETCDF.

The UM provides two (legacy) initialisation routines which are dedicated to initialising JULES. These were added to the JULES interface but are not used by the stand alone wrapper. Similarly the standalone wrapper has its own initialisation routines which were added to JULES but are not used by the UM. One reason for adding these to JULES is that they make use of a lot of internal JULES data

f) Initialisation Issues

As mentioned in the previous section, the UM has its own initialisation routines. The stand-alone JULES wrapper also has its own initialisation routines and both have been added to the JULES interface. The new JULES initialisation routine mentioned earlier, which reads in the JULES input data, is also required. Further, this initialisation routine needs to be split into two (called init1 and init2) as the wrapper needs to allocate any required data before JULES reads in the data, and the wrapper can not do this until is knows what the data sizes are, and the data sizes are received from JULES. For JULES in the UM this results in the following initialisation routines being called

INIT1

LAND_SURF_INIT

INIT2

INIT_VEG

The situation gets worse with the addition of new science and, for example, another initialisation routine (init3) needed to be added to support ED[17] within JULES. There are a number of ordering dependencies in the above routines which make life difficult and required some code modification.

What is required, but has not yet been implemented, is a rationalisation of the init routines into two: one which reads any initial configuration information and a second which reads any required input data, however this rationalisation will require some modifications to the UM,

[17] Ecosystem Demography (ED) is a dynamic vegetation model.

and is left as future work.

g) Integrating JULES and the UM Land Surface Scheme

A separate Met Office project has integrated the science of the JULES model and the UM Land Surface model into a new version of JULES. It is expected that the modularisation effort described here will be migrated to this integrated JULES model (in a future project).

Summary

Despite the complexity of JULES, component-compliance was not difficult to implement as the interface of JULES is already a set of subroutines with most of it's data being passed by argument. The only real issue was the removal of data sharing using COMMON or Modules. However, the restructuring of such a large code to make it more modular and more easy to work with is a much larger job, which also requires restructuring of the host model (the UM) and this work is ongoing.

9.3 Climate and Integrated Assessment Modelling

Introduction

Coupling Systems that target the Earth System Modelling (ESM) domain provide support for the particular characteristics that this domain requires, such as scalability to large numbers of cores and regridding transformations, for example. It would be beneficial to users if the same coupling system were also able to support the requirements of related domains, particularly in the situation where model codes may be shared between the domains. One particular example domain of increasing importance to ESM is that of Integrated Assessment Modelling (IAM); this domain was recognised as such in the recent ISENES Strategy document.

IAM links ESM models (often Climate models, or even emulators of Climate models), land-use models and economics models together into integrated (i.e. coupled) models in order to help inform policy makers about the choices they need to make to implement certain policy scenarios - such as achieving a maximum 2 degree warming by 'managing carbon emissions, for example. Such models can also be used to play 'what-if' games so that policy makers can investigate what effects their policies might have.

In this Section some of the differences in requirements between IAM and ESM are outlined and the question of how these requirements relate to the functionality required in a coupling system is discussed [18]. Finally, an example of how BFG is being used in the IAM domain is presented. This example is fully supported in the current version of BFG2. The example highlights the extensions to BFG that have made this possible. (This example has benefited from the work undertaken in IS-ENES on program- and component-compliance. Specific extensions for the support of multiple languages and the development of the example itself has been funded under the EU ERMITAGE project).

Differences between IAM and ESM

To date, the individual models that are used in IAMs have typically been developed as standalone codes with little consideration given to their potential for coupling with other codes. This observation implies that the aggregation approach to coupled modelling in IAM is most appropriate. The models are also typically much smaller (in terms of lines of code) than typical ESM models and they are not usually internally parallel.

[18] Some further discussion of this topic can be found in Chapters 8 and 9, (on the CIAS IAM and the Summary, respectively), in the book “Earth System Modelling – Volume 5, Tools for Configuring, Building and Running Models”, Eds: Rupert Ford, Graham Riley, series eds: Reinhard Budich, Rene Redler. Springer Briefs in Earth Sciences, 2012.

In IAM, uncertainty analysis is routinely performed and this typically involves a large number of runs of a particular coupled IAM model. This means that individual coupled models are typically expected to run in a matter of hours, or less (although this is not always the case). Emulation is one solution that is sometimes used to ensure that models run in acceptable time.

Whilst the performance of a coupled model is always an issue, it is of less importance in IAM than ESM, and distributed coupled implementations, possibly across firewalls, may be considered. One reason to allow a distributed coupled solution is that licensing issues can be respected; in IAM many model codes have licenses which restrict access to source and binary code.

Unlike ESM, where Fortran is dominant, models in IAM can be written in a number of languages. Economics models are typically written in GAMS, emulators are typically written in R, and more ESM-like models are typically written in Fortran, C or Java.

Finally, whilst ESM models are almost exclusively time-stepping models, in IAM some interactions between models may be convergence-based. For example, two models may continue exchanging data until they iterate to convergence on a particular solution. Coupling scenarios may involve a combination of time-stepping and convergence-based iteration.

Coupling System Requirements

The different requirements that IAM has compared with ESM leads to a number of different and additional requirements for an underlying coupling system designed to support ESM only. These are described next.

IAM coupled models may require more geographically distributed solutions. Distributed MPI can, to some extent, provide the required functionality, but there may be cases when a more distributed solution, such as one based on Web Services, would be beneficial. Whilst distributed solutions are not usually expected in ESM, it has been recognised that web services might also be beneficial in the future, particularly for community access to ESM models, and to provide the ability to include coupled models into more general Workflows. In particular, ESMF have recognised the need for this and ESMF provides basic support for the wrapping of models as web services. BFG is designed to be able to add new 'target' communications technologies as needed and a prototype web services solution was provided in BFG1. BFG2 could be extended to support coupling via Web Services directly, or to export a particular model (or set of models in a composite) as a Web Service. A Masters student is currently working to develop this solution.

IAM models are written in a number of different languages. These include Fortran, C, Java, GAMS, and R. ESM coupling systems universally support Fortran, many also support models that are written in C and some are developing, or have developed Python APIs (for example, this is the case for ESMF and MCT, the US coupling systems). Language interfaces to other languages are only viable if a language supports them in a suitable way. This is not the case for models written in GAMS. The solution taken by BFG (in the EU-funded ERMITAGE IAM project) is to support Fortran, C and Python interfaces in the first instance (and possibly to add support for R and Java in the future). The Python interface allows users to write python wrappers around their GAMS and R code. This wrapper provides all input coupling data, runs the code and extracts the required output coupling data. The input and output is typically done via files, and the model is "exec"ed[19] by the Python code. Python was chosen as the language to do this as it provides the required functionality in a simple way and, was, in fact, already being used in this way to manually couple some IAM models together.

IAM coupled models are amenable to (geographically) distributed solutions. Further, individual models in IAM are typically written independently and, in the case of GAMS, there

[19] "Exec"ing is to launch an executable from another.

is no obvious language interface API that can be used. Therefore, the typical approach adopted when coupling is for minimal intervention in the code. To put it another way, there is little benefit in modularising model code if a tightly coupled solution is not required. BFG2 supports minimal intervention in code through the use of its program compliance model or by the wrapping of existing executables in a supported language (and, typically, the wrapping language is expected to be Python since it provides good support for modifying the file-based data which is used in coupling exchanges). OASIS3 supports a similar API to BFG's program compliance for the input and output of coupling data.

IAM couplings are not necessarily timestep based. One reason for this is that Economics models often provide solution data at intervals over a period of time (i.e. they provide a time series as output) and, therefore, time series information has to be passed between these models and others. One way in which to obtain consistency between two models, where at least one of them provides or expects time series information is to iterate to convergence. In contrast, some models (typically land-use and climate models) can couple between (or even during) timesteps. In BFG2 the solution to the requirement for convergence-based behaviour was to extend the schedule, defined in the BFG metadata, to support convergence-based iteration as well as timestep-based iteration. The stopping criterea for a convergence loop is provided by a coupling between a model (or transformation) and the convergence control loop. Thus, in BFG2 the, very restricted, logical concept of a control “model” has been introduced. A control model has an input for each convergence loop specified in the schedule. Models (or transformations) may pass logical information to these convergence loops to indicate whether the loop should continue or terminate. The authors are not aware of convergence solutions in other coupling systems in ESM apart from PALM[20], which supports more general communication between a defined schedule and a model.

Coupled GEMINI-E3/GENIE-EM Example

The example provided below is one that has been developed in the EU ERMITAGE project. The ERMITAGE project is attempting to link together Climate, Land-use and Economics models in news ways from both a scientific and technical (using BFG2) perspective.

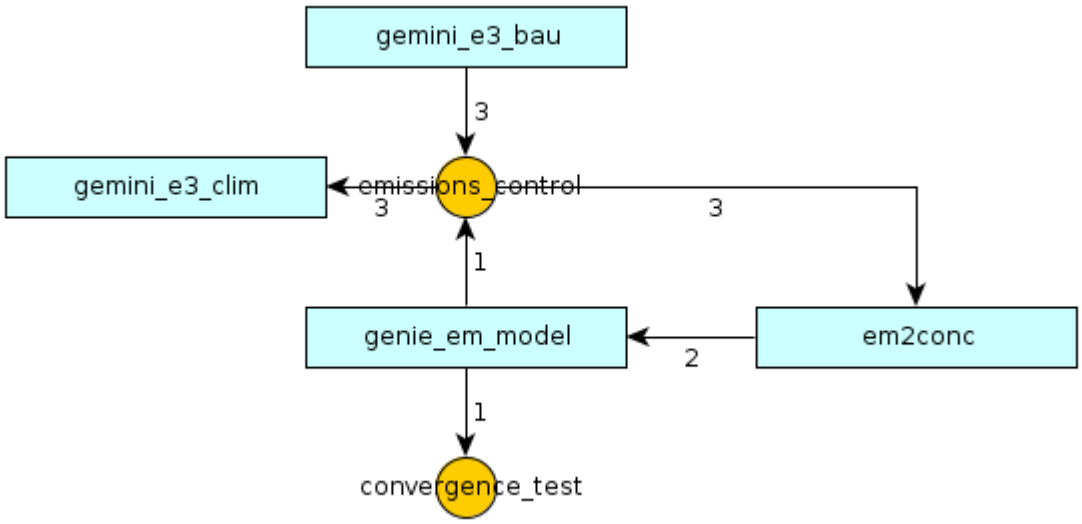
The project is working towards fully coupled IAM's by first looking at the links between pairs of models. The example provided here demonstrates one aspect of a IAM - the link between the Economics and the Climate.

The Economics model is called GEMINI-E3 and is written in GAMS. A BFG component-compliant python wrapper has been written for this model so that it can couple with other models.

The Climate model GENIE-em is an emulation of the GENIE paleoclimate model and is written in R. Again, a BFG component-compliant python wrapper has been written for this model so that is can couple to other models.

The diagram below summarises the models (represented as (blue) rectangles) and transformations (represented as (yellow) circles) used in the coupled model as well as the connectivity between the models (represented as arrows between models and/or transformations - the number of types of data exchanged is represented by the number on the arrow). This diagram was generated directly from the BFG2 composition metadata, which describes the connectivity of the coupled model, using a BFG2 utiltity program.

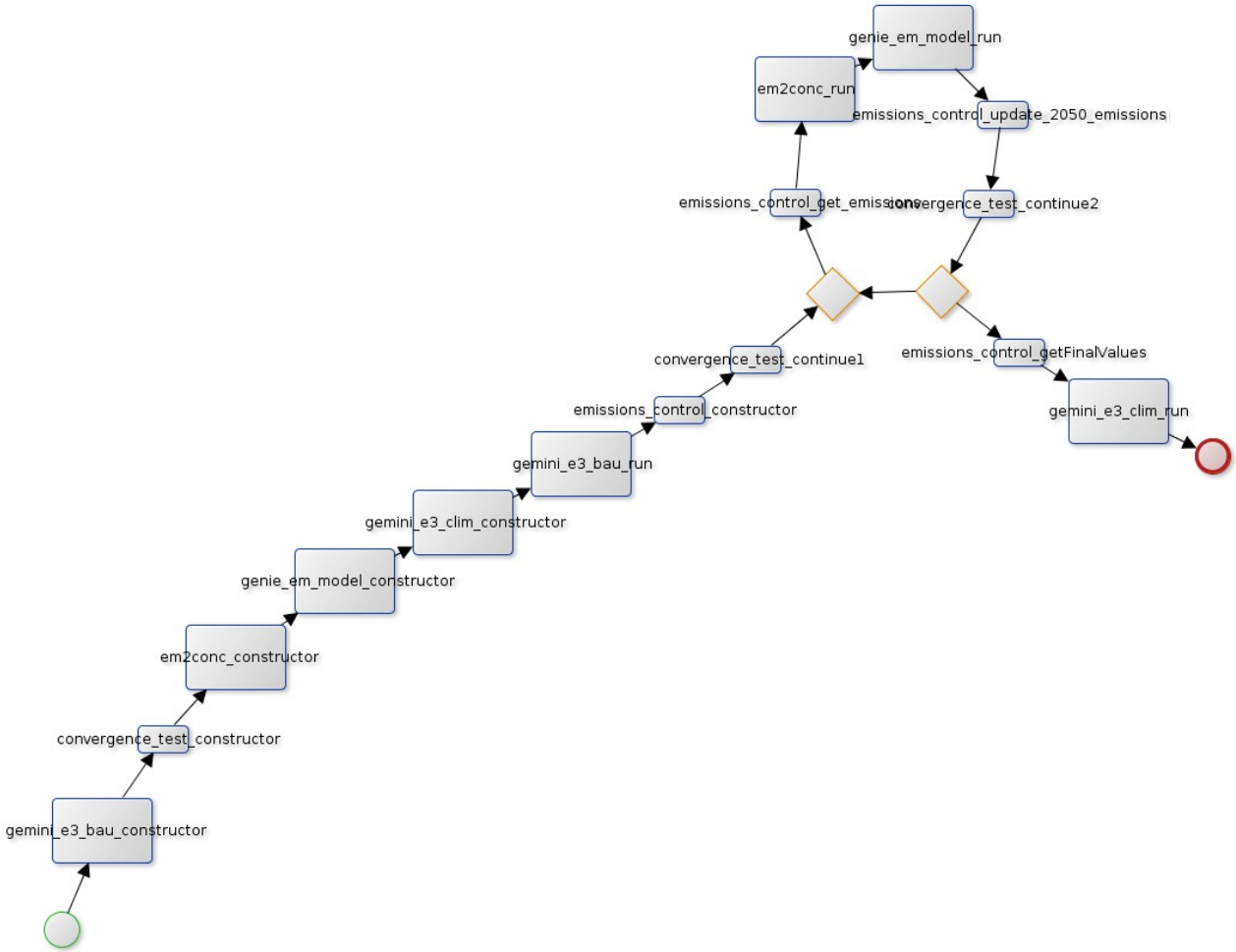
[20] See http://www.cerfacs.fr/globc/PALM_WEB/index.html



The coupled model begins with a “business as usual” execution of the GEMINI-E3 economics model (gemini_e3_bau). This produces a baseline emissions timeseries (for CO2, NO2 and CH4) for the years 2005 to 2050 which are provided to the Emissions Control (emissions_control) transformation. The coupled model then iterates, calling, in turn, the GENIE emulator (genie_em_model), the Emissions Control transformation and an Emissions to Concentrations model (em2conc) until a profile is produced which is consistent with the climatology in the GENIE emulator. Lastly, this consistent profile is passed to the “climate” flavour of GEMINI-E3 which computes the final profiles.

The Convergence transformation (convergence_test in the diagram) implements the stopping criteria for the loop. It returns a boolean value to indicate whether the loop should continue or not. This value is provided to the BFG2 control loop which is not represented in the above diagram since the control loop is considered to be a virtual model.

The following Figure presents the coupled model schedule that has been defined using the Business Process Modelling Notation (BPMN). This diagram is generated from the BFG2 deployment metadata using a BFG2 utility program. In this case, the different entrypoints for each of the models are shown. Entrypoints of the scientific models are shown as larger (blue-edged) rectangular boxes and transformations as smaller (blue-edged) rectangular boxes. The names in the boxes are the concatenation of the model name and the entrypoint name for the particular entrypoint. The light (green edged) circle indicates the start point for model execution and the bold (red-edged) circle indicates the termination point. The control flow is shown by the directional arrows and the diamond shaped (yellow) boxes represent the start and end of (iteration or convergence) loops in the schedule.



In this schedule the majority of the model constructors are called first (these are treated as a special type of entrypoint in BFG2). The GEMINI-E3 “business-as-usual” model is then called. The output of this model is actually passed to the constructor of the the Emissions Control transformation, hence the constructor for this transformation is called after the GEMINI-E3 model has run. An initial convergence test entrypoint is then called before the model enters the convergence loop, just in case the convergence criteria has already been met. If the criteria has not been met (the normal case), the convergence loop proper is then entered and this loop continues until the second convergence test reports False (so strictly, this test is a continuation test, rather than a convergence test). Once the loop has completed, the final emission values are provided to the GEMINI-E3 Climate model by the getFinalValues entrypoint of the emissions control transformation and the program subsequently terminates.

In the above case, all of the models are run in a single executable, passing data via argument passing. However, a simple change to the deployment metadata can be made to request the use of MPI, OASIS3, OASIS4 or ESMF to manage the exchange of coupling data. In the future, a Web Services 'target' infrastructure will be made available. Using other facilities in BFG2, these models could also be exported as OASIS3, OASIS4 or ESMF-compliant models as discussed in the first example in this section.

10. Conclusion and future work

The aim of the research and development of BFG2 in IS-ENES is to demonstrate that a coupling technology that would support both program-based coupling of model codes (through the use of in-place put and get-style operations) and component-style composition is a viable option. The generative programming techniques that can underpin the approach have been prototyped and demonstrated in simple example coupled models. The key to the approach is to support the separation of science code from that of the coupling infrastructure. Enabling both coupling approaches within a single framework gives users the ability to choose the approach appropriate to their needs. Further, supporting the isolation of model science code from the coupling technology has been demonstrated to promote the engagement of external communities in coupled model development and in the sharing of models across and between communities. This is exemplified in the case of the Integrated Assessment community in the EU Ermitage project and in the Tyndall Centre's CIAS tool.

Re-factoring the BFG code based on the templates approach has made it much easier to maintain and extend and plans to release BFG under an open-source licence are well advanced. Other work in IS-ENES has seen BFG extended to support languages other than Fortran (including C and Python). BFG tools can also now be accessed on-line via the BFG portal and examples of coupling metadata are available for browsing.

Future work on BFG includes plans to improve and extend the current basic support for OASIS3 and ESMF as target coupling technologies and also to increase support for parallel models (and support the definition of parallel partitions). As a larger-scale example, the plan is to develop a program-compliant BFG2 version of the UM atmosphere model and demonstrate exporting it to OASIS3, OASIS-MCT and also to ESMF (and possibly OpenMI) in order to illustrate how the BFG2 approach can be used to couple a model using a variety of existing coupling technologies and where the developer need only make minimal changes to metadata (and re-run the BFG code-generation phase) but with no impact on the scienfic code in a model.