IS-ENES – WP8

D8.5 - Towards Flexible Construction of ESMs using BFG, Final Report

Abstract:

Grant Agreement Number:	228203	Proposal Number	FP7-INFRA-2008-1.1.2.21
Project Acronym:	IS-ENES		
Project Co-ordinator:	Dr Sylvie JOUSSAUME		

Document Title	Towards Flexible Construction of ESMs using BFG, Final Report	**Deliverable**	D8.5		
Document Id N°	ISENES_D11.5_20130227.doc	**Version**	1	**Date**	27/2/2013
Status	Final				

Filename	ISENES_D8.5_Towards_Flexible_Construction_of_ESMs_using _BFG_FINAL_V1.0

Project Classification	Public

Authors

G.D. Riley, P. Slavin, R. W. Ford (now at STFC Daresbury)   **UNIMAN (6)**

Revision Table

Version	Date	Comments	Authors, contributors, reviewers
0.1	08/01/13	First draft	P. Slavin, G.D. Riley
0.2	25/01/13	Added Infrastructure API details	P. Slavin, G.D. Riley
0,3	08/02/13	Extend coverage and edit	G.D. Riley
0,4	13/02/13	Ready for project review	G.D. Riley
0,5	27/02/13	Responses to reviewer comments and final tidying	G.D. Riley
1	26/03/13	Responses to final reviewer comments and release of Version 1.0.	G.D. Riley

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, and other software infrastructure, as provided by technologies such as OASIS, MCT and ESMF.

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

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

The recommended 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.

1.1.　Overview

This document presents developments in the last period of the IS-ENES project related to research in flexible coupled modelling and the Bespoke Framework Generator (BFG). This work builds on that reported in D8.3, "Towards Flexible Construction of ESMs using BFG", and background on BFG presented in that deliverable is not repeated in full here. Familiarity with Sections 2 and 3 of D8.3 is assumed.

The developments described fall into the following categories:

1. Investigation of extensions to BFG to provide support for parallel models, utilising, where possible, the functionality provided by existing software infrastructures and coupling technologies, such as MCT, ESMF and OASIS (OASIS3-MCT);

2. The provision of support for parallel models using the emerging quasi-uniform grids (for example, cubed sphere and icosahedral) which are designed to extend the scalability of models, as required for future, many-core computing systems;

3. Extensions to BFG to support the integrated assessment community – an update to work reported in the previous deliverable, D8.3;

4. Further developing the BFG portal.

The work in items 1 and 2 will be included in a future release of BFG; BFG2 remains the current release.

As described in the previous deliverable, the philosophy of BFG has two main themes. The first theme is the isolation of science code from the technical details of any specific supporting software infrastructure technology. The second theme is that of metadata-driven configuration and code-generation.

Isolating science code from details of specific infrastructure technologies provides flexibility in the use of that model, both in terms of coupling a model to other models and in terms of deploying models onto a variety of computing systems. For example, it allows the implementation of the infrastructure to change with no impact on the science code. This

promotes, among other things, the sharing of models between groups in the community by allowing one model to be used in the context of a coupled model implemented explicitly using another technology. For example, such a model may be 'targetted' to use ESMF or OASIS as the coupling technology as required.

As described in D8.3, BFG takes in metadata, in the form of XML files, describing:

1. the individual models involved in a coupled model,

2. the composition of the models into a coupled model; providing information concerning the data to be exchanged between each model, including coupling rates etc.,

3. deployment requirements such as the mapping of models to executables and their execution orders (models may be executed in-sequence or concurrently, for example. Deployment metadata also specifies the target coupling technology to be used to implement coupling exchanges between models.

Using the metadata, BFG generates the additional code required, beyond the source code of individual models, to implement the coupled model. The generated code consists of one or more main program control codes (an implementation of what is often called the Driver layer or Superstructure) and communication library code (often called infrastructure code) to implement the exchange of coupling data between models. Both the driver-layer code and communications-layer code make use of the selected target (pre-existing) coupling technology.

Achieving isolation of science code requires the definition and implementation of an Application Programmer Interface (API). As described in D8.3, BFG supports two levels of interfaces: a minimal interface, known as the **component-compliant API**, for scientific model code implemented essentially as subroutines, and a less flexible **program-compliant API** for use with models implemented as existing programs.

In pursuit of providing support for parallel models, the API provided to the scientific model developer has to be extended beyond providing support for the coupling of parallel models to include, for example, access to functionality supporting halo-exchanges within a parallel model.

Several existing infrastructures provide support for such functionality, including MCT and ESMF, and several modelling groups simply provide their own infrastructure library for this functionality; on the other hand, OASIS only provides support for coupling, for example. Given that an aim of BFG is to provide the flexibility for a coupled model to use any of a number of existing ('targeted') software technologies, the aim in the design of the API to be used by the scientific developer discussed in this document is to provide a minimal specification of the functionality required (e.g. for the coupling of parallel models and for halo-exchanges within parallel models). The rest of this section gives an overview of the API and its implementation and use.

The translation of the API call into an implementation in a specific technology (say, MCT or ESMF for halo-exchange, or ESMF or OASIS3-MCT for coupling) is achieved in the implementation of the API with the support of metadata. Prior to running a coupled model, in addition to the standard BFG metadata required to describe models, their composition and deployment, metadata describing the parallel partition of computational meshes is required. Further, in addition to selecting a 'target' coupling technology, the target technology to be used for halo-exchange etc. must be specified.

In an initialisation phase for the coupled model, the specific mechanisms required to implement halo-exchanges (of the required halo depth) and coupling exchanges are set up For example, for MCT, appropriate global segment maps, rearrangers and routers etc. must be created and registered for each coupling field ready for use by models; or ESMF routers must be initialised; or OASIS3-MCT definition calls must be invoked and appropriate

namcouple file generated, etc.. Details of the mechanisms created (for example, pointers to the segment maps, routers etc. to be used with each field in the models) must be registered in model-specific data structures for later use by scientific models.

At run-time, an API call made by a model to perform a halo-exchange or a coupling (or some other infrastructure functionality) is linked to the specific implementation mechanism (defined through metadata and implemented and registered in the initialisation phase) through the use of a standard BFG 'tag'. The tag provides a key to the accessing of model-specific data structures which contain details of the implementation mechanism to use in a specific context. For example, the tag allows a model to access the pointer (in the model-specific data structure) to the correct MCT rearranger to use for the halo exchange of a particular field with whatever halo depth was specified.

This document presents background details of the provision of support to a parallel model in using (partitioned) non-uniform meshes. The main part of the document concentrates on the use of MCT to provide implementation of the required functionality, with the focus on the provision of (simple) halo-exchange within the processes of a parallel model and on the coupling exchange between two models of a scientific 'field' defined on a non-uniform mesh which is partitioned differently in the two models.

The complexity of providing grid transformations between different meshes in the two models is avoided. The emphasis instead being on the provision of an **Infrastructure API** that can be implemented in a variety of existing technologies. In this approach, the existing technologies, such as MCT, ESMF, OASIS3-MCT etc., are viewed as providing a 'tool-kit' of functionality. Through the use of the Infrastructure API and with the use of metadata, a coupled model developer may select the 'best-of-breed' existing implementation technology appropriate for their circumstances.

In this document, the term 'Infrastructure API' will be used to refer to both the set of operations exposed to users by the API and to the coded implementation of the API functionality.

It is hoped that the approach of isolating science code from specific infrastructure technology implementation details, and the flexibility to select appropriate technology, may also inform the current debates in the ESM community concerning framework interoperability and the possible adoption of a single, common software infrastructure. For the latter initiative, see the recent report, "A National Strategy for Advancing Climate Modeling", from the National Academy of Sciences in the U.S, 2012 (https://download.nap.edu/catalog.php?record_id=13430).

1.2.   Implementation status

The work described in this document has resulted in a manually generated prototype example of what a BFG-generated solution would look like for a coupled system consisting of parallel models (this is a so-called 'manUgen' version). This work has not yet been integrated into a new BFG release and BFG2 continues to be the latest release. The prototype targets MCT as the existing infrastructure. Separately, investigations of the use of OASIS3-MCT and ESMF in supporting parallel models with quasi-uniform meshes have been undertaken. These exercises support the assertion that the layered, metadata-driven Infrastructure API described in this document will allow the incorporation of both OASIS3-MCT and ESMF as BFG 'targets' for coupled parallel models in future work resulting in a new release of BFG

A snapshot of the prototype code is available on the BFG website, http://cnc.cs.man.ac.uk/projects/bfg

1.3.　The cost of BFG-compliance

BFG is a metadata-driven code-generation system. Making a model BFG-compliant requires metadata describing the model to be generated and, potentially, changes to the model code. The following comparison with making a model either OASIS-compliant or ESMF-compliant can be made.

A model that exists in its own program unit (separate executable) can be made OASIS-compliant through the addition of code (calls to oasis library routines) to define the grid and partition used and to define the coupling variables. In addition, put and get calls must be added to the code and an appropriate namcouple file written to control the coupling.

To make the same model BFG *program*-compliant requires metadata describing the model (and its composition into a coupled model) to be written but the only code changes required are the planting of calls to BFG puts and gets, equivalent to the OASIS puts and gets, a call to a BFG initialisation routine and calls to BFG routines that mark significant control phases, such as the end-of-timestep. BFG will generate the appropriate grid, partition, and variable definitions for OASIS – these will be contained in the BFG-generated initialisation routine – and an appropriate namcouple file.

Making a model BFG *component*-compliant is a similar process to making the same model ESMF-compliant. Again, BFG metadata must be provided, and from the metadata, calls to ESMF initialisation routines are generated, so do not have to be written. The model's (init, run and finalise) code is essentially the same, though the coupling data in the BFG model will be explicit in the argument list, and BFG-generated wrapper code will contain the required import and export states and manage them. BFG will also generate appropriate ESMF coupler components, again, based on the metadata description of the models and coupling requirements (i.e. BFG composition and deployment metadata).

In summary, BFG compliance requires the generation of metadata and this results in a (potentially significant) reduction in the amount of code related to coupling that needs to be hand-written in individual models and in the coupled model.

1.4.　Organisation

This document is organized as follows; Section 4provides an overview of the motivation for the work reported; Section 7describes the Infrastructure API for which a prototype, manually generated implementation has been developed; Section 14details the implementation of the Infrastructure API and provides some examples of its use by scientific models; Section 19discusses the implementation of the Infrastructure API in terms of a specific, existing target technology, the Model Coupling Toolkit (MCT); Section 22briefly discusses the targetting of OASIS3-MCT and ESMF; The next two sections provide updates on two other BFG-related topics discussed in D8.3: Section 23provides an update on the online access of BFG and related tooling and Section 24discusses developments in the use of BFG for Climate and Integrated Assessment Modelling. Finally, Section 24concludes and offers some pointers to future work.

2.　Infrastructure API
2.1.　Introduction

This section examines the structure of the Infrastructure API and considers the design decisions taken in arriving at this structure. Comparisons are made with the decisions made in the current BFG release, BFG2. The guiding principle in making these decisions has been the aim to provide a useful and convenient range of functionality to scientific model developers, but to do so in a manner which provides flexibility in the choice of an existing

software infrastructure to implement the functionality. As such, a distinction has been maintained between the *mechanisms* (i.e. the functionality provided by the API) that are made available to scientific model developers, and the *policies* which determine how these mechanisms are used in a particular model to implement the modeller's design. Whereas the mechanisms are defined by the Infrastructure API, models retain the freedom to make use of the API in the manner that is appropriate to achieve their scientific objective. Thus, design decisions relating to the internal data structures and algorithms to be used in a model remain the preserve of the model developer, with the API providing routines which are sufficiently general in their application to act as tools to be used by model developers (a so-called 'tool-kit' approach).

In keeping with this objective, the Infrastructure API is implemented as a set of modular layers, permitting implementation choices, particularly in terms of the choice of using a particular existing technology, to be made according to the requirements of the coupled model being developed and the computing system to be used, etc. while maintaining a consistent interface to scientific models.

This modular structure also enables the simple integration of scientific models into an automatic coupling framework, such as the Bespoke Framework Generator (BFG). Interactions between models then take the form of invocations of API procedures having well-defined interfaces that remain constant in spite of any changes to the implementation of the underlying implementation layer.

The resulting design is characterised by a loose coupling between the conceptual layers describing the structure of a coupled model. This structure is depicted in Figure 1, which is explained in the next section, where the hierarchy of layers comprising a coupled mode is shown to be sub-divided into a range of alternate implementation-specific modules, which may be substituted without implications for the model code.
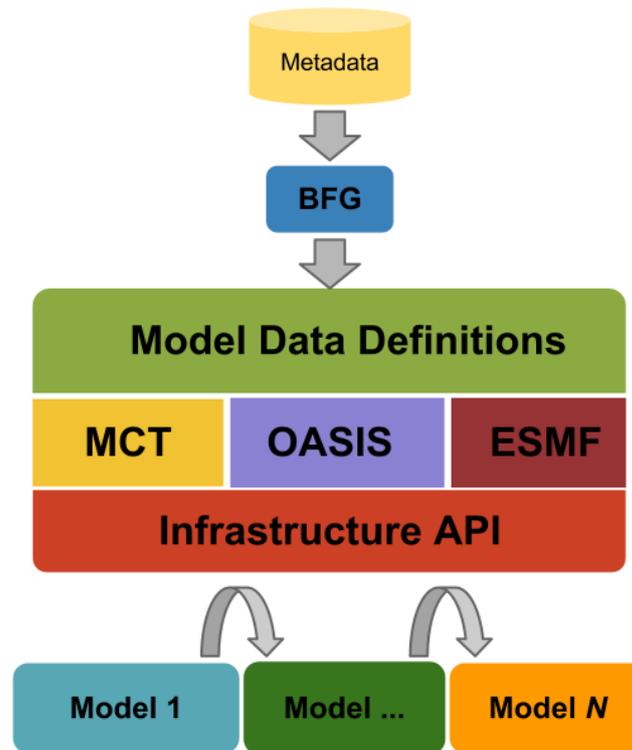
Figure 1: Modular structure of the Infrastructure API and Metadata Framework showing how BFG metadata flows through the BFG system resulting in Model Data Definitions which configure the run-time use of existing coupling technologies, such as MCT, OASIS and ESMF. The technologies are made available to the models in a coupled model through the infrastructure API layer.

2.2.   High-Level Structure

Models are isolated from implementation details through the use of the Infrastructure API module depicted in Figure 1. Models then perform their data exchanges and associated operations by calls to the relevant API procedures. Implementation of these exchanges by the API code makes use of lower-level coupling, communications and other infrastructure functionality, the details of which are hidden from model code. As such, a consistent high-level interface is presented to models by the Infrastructure API, irrespective of the particular infrastructure library employed. This latter library may, therefore, be substituted for others in a transparent way from the models' perspective, according to the requirements of the execution environment. The Model Data Definitions layer is created by BFG from the metadata associated with the coupled model and, for example, captures the coupling requirements of the model in a coded fashion. The model data definitions are added to at run time as specific implementation choices for coupling exchanges and halo exchanges are determined. For example, given the specification of a particular coupling technology, say MCT, in the BFG metadata, for each coupling exchange specified in the composition metadata, appropriate Global Segment Maps and MCT routers must be defined and set up for the model partitions specified. Pointers to these maps and routers will be saved in the

appropriate model-specific data structures so that the correct router can be used when the coupling exchange is invoked by a model. Further details of the population and use of this information can be found in Section 15.

This technique is implemented by a set of subroutines which accept parameters expressed in terms of the client model's semantics and which then convert these parameters into the specific forms required by the underlying implementation layer.

The operations provided by the Infrastructure API can be divided into three principle categories:

i.   Type conversion operations

ii.   Infrastructure functionality specification operations

iii.   Communication and other operations (including, for example, coupling exchanges and halo-exchanges)

The first category of operations permits the data types and structures native to a client model to be expressed in a form suitable for use by the underlying communications layer. As such, the client model can preserve the forms and types of its data in the manner most appropriate to its internal operations, while interacting with the Infrastructure API by way of data types which remain unchanged irrespective of the internal implementation of the infrastructure layer.

This is achieved by way of an internal translation layer in the Infrastructure API which presents a uniform interface to client scientific models from which it receives model-specific parameters that are used to construct (during initialisation) and access (at model run-time) an instance of a structure which encapsulates that model's state. Further details of the implementation of these operations are given below.

The second category of operations consists of infrastructure specification operations. During initialisation (both of individual models and the coupled model), API routines may be called in order to define the relationships (for example, requirements for halo exchanges and coupling exchanges) in and between models that are to be implemented by the Infrastructure API. For example, in a model, partition information for the mesh(es) used by the model may be defined and, based on the partition information, halo exchanges with appropriate halo depth may be set up, and coupling exchanges with other models specified. This process is a process of configuration driven by metadata describing the model, mesh partitions and halo and coupling requirements for each field in the model. This is the process illustrated in Figure 1. In response to these definitions, the Infrastructure API translates the representation of the relationships, as specified by the models, into specific forms that employ the syntax of the appropriate infrastructure implementation layer(s), for example, MCT or ESMF for halo-exchange and coupling, OASIS3-MCT for coupling. These routines will set up appropriate re-arrangers and routers for MCT, or initialise router handles in ESMF or define the partition and variable definitions required by OASIS3-MCT. In summary, BFG metadata is processed (by a BFG code generator) resulting in the generation (amongst other things) of model-specific data structures which capture the requirements for each model for functionality such as halo exchanges and coupling exchanges. During model (and coupled model) initialisation, these data structures are read, and appropriate Infrastructure API routines invoked to initialises the selected coupling technology mechanisms to implement the halo and coupling exchanges . Information about the mechanisms set up in this process is added to the model-specific data structures (such as pointers to MCT re-arrangers for halo exchanges etc.). This information is then available for access in Infrastructure API calls made in the 'run' routines of models to initiate actual halo and coupling exchanges.

The final category of facilities provided by the Infrastructure API is concerned with operations

that are invoked as the models run. These routines trigger the communication, exchange or transformation of data within or between client models. For example, routines exist to effect halo exchange within a model, to enquire about the storage of data on other processes and to communicate data between coupled models. As with the other categories of facility, client models invoke these operations using a high-level representation of the operation to be performed (in terms of the semantics of particular models) and the detail of effecting the operations is determined by the Infrastructure API using the selected implementation.

The theme which pervades each of these categories is the presentation by the Infrastructure API to the models of a uniform and consistent interface which encapsulates the attributes of the target, existing infrastructure-layer implementations. The remainder of this section describes some of the facilities that are offered to client models and gives details of their implementation.

2.3.   Representation of Model Entities - Data Structures and Types

2.3.1.   Partitions

In order to support an intuitive (to the model developer) representation of the data structures and types employed by client models, the Infrastructure API defines several generic types which correspond to commonly used structures within scientific models. While the naming and functionality of these types indicates their intended usage, each type is defined in a sufficiently abstract manner that few assumptions are made about the detail of its use within a model. That is, the Infrastructure API provides a flexible mechanism for models to employ, but does not prescribe the how they are to be used in a model. This section describes support for models which use unstructured meshes (including quasi-uniform meshes). These meshes are being incorporated in the new dynamical core models being developed by most ESM groups around the world as they promise better scalability.

The use data types that are populated at run-time, rather than being generated from (static) metadata by BFG, is a departure from the traditional approach taken by BFG.

Perhaps the most fundamental type defined in the Infrastructure API is the Segment. This is defined (Figure 2) as a primitive which is intended to form a linked-list structure that

```
type Segment

    integer                      ::base, run          type(Segment),pointer
    ::next

end type Segment
```

Figure 2: The generic Segment *type*

describes the decomposition of a computational mesh into a set of partitions comprised of regions of contiguous global indices. This is the de-facto standard for the representation of partitions of an unstructured mesh and is used by MCT, ESMF and OASIS. Each Segment type describes one such region, specifying the starting index in its base value and the number of contiguous indices in this region in its run value. The next pointer connects the region to the remaining regions which collectively describe a complete partition of the mesh.

This structure constitutes a high-level representation of the decomposition of any generic memory structure in which component elements may be identified by way of an global integer index value. As such, the type is equally suited to the representation of a piecewise

partition of a matrix as to the representation of a partitioned computational mesh. Similarly, a sparse structure may be represented by defining Segments which correspond to only the populated regions. Within the Infrastructure API, and in the examples that follow, this type will be employed to represent both the elements of a mesh, and the vertices (and edges) of which those elements are comprised. In scientific models, fields may be associated with each of these topological entities (elements, vertices and edges) depending on the discretization used in the numerical scheme employed. The term *grid* is usually used to describe a mesh and its discretization in the ESM community, especially for finite-difference models. The finite-element community usually do not refer to a grid but only to a (computational) mesh and the discretization used. The collective structure which arises from the linking of several Segment types is referred to as a Segment Map.

In a parallel model, each process associated with a model with call Infrastructure API routines to define its own local segment map. Within the Infrastructure API layer, a global segment map view is maintained by each of the target infrastructures (MCT, ESMF, OASIS). In MCT, this global view has to be constructed through the use of calls to lower-level routines. ESMF and OASIS have high-level routines which automatically construct the global view.

2.3.2. Local and Halo storage

The Infrastructure API can read mesh decompositions of the sort that are produced by the SCOTCH or METIS mesh partitioning systems and translate these into a Segment-based format.

Having defined a (global) segment-based view of a mesh partition, other Infrastructure API routines can be called to construct halo definitions of arbitrary depth. Halos will typically be required for fields defined on elements, edges and vertices.

This stage of the use of the Infrastructure API typically commences with the reading of a mesh partition (either from a file or set of files or as the result of a previous phase of computation at run-time) and results in the construction, for each process associated with each model, of two sets of mesh points, the local and halo sets. Each of these sets are represented internally by way of the Segment structure described above. In addition, a range of routines are available to enable models to test the ownership of a mesh point, manipulate local and halo points as sets, and exchange halo points between the processes of a model. From the perspective of a client model, the set of local or halo points may be considered as an abstract unit and manipulated using routines which maintain this abstraction (see Figure 3).
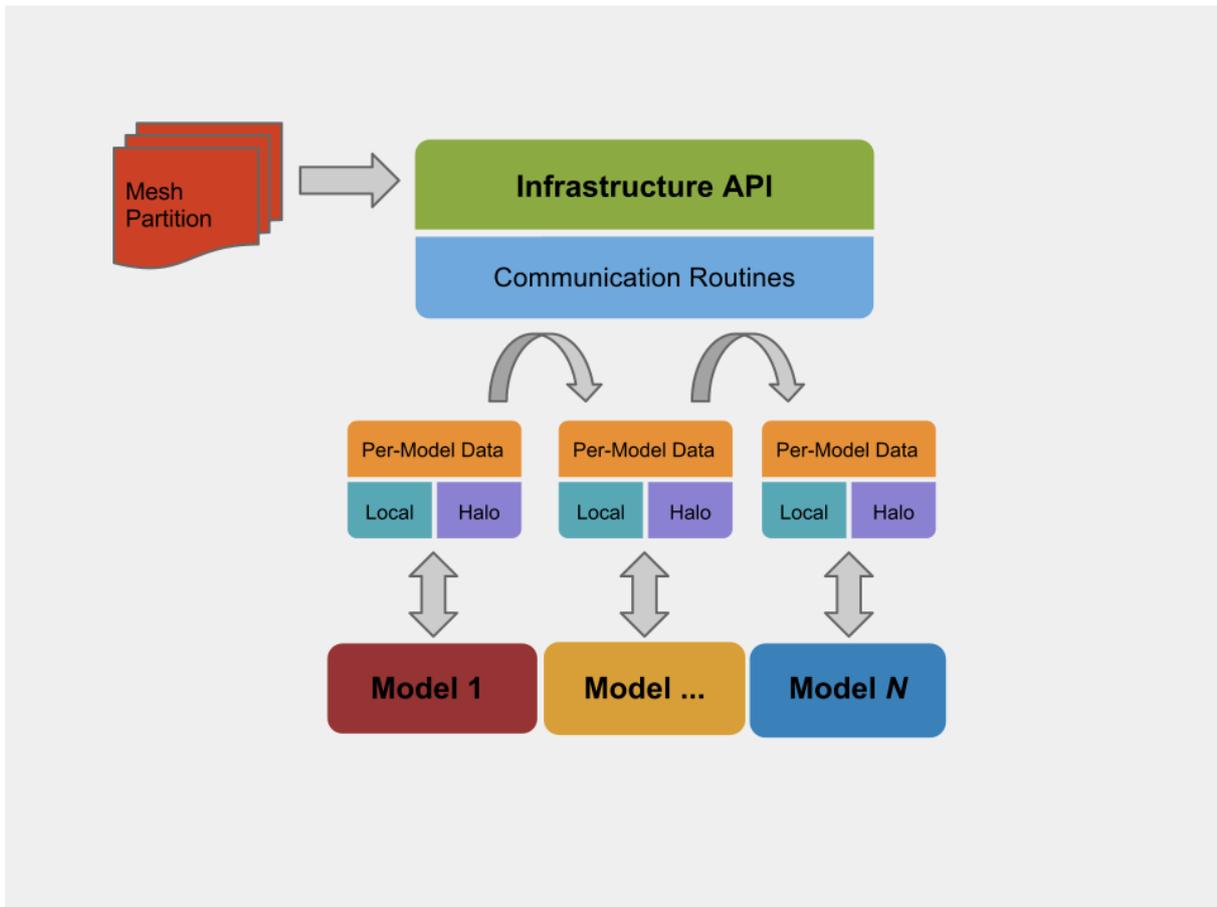
Figure 3: Mesh decomposition into per-process Local and Halo data

As an example, it is possible that two models coupled using the Infrastructure API could use the same mesh, but each employ a different decomposition for their local calculations. In this case, the set of all processes participating in each model possesses a complete partition of the mesh, and to support coupling between the models, the infrastructure has to be made aware of the correspondence between the partitions (across processes) used by both models. This correspondence is set up during the model and coupled model initialisation, driven by the requirements described by the BFG metadata, as described in Section 9.Further, a single model may have to deal with more than one mesh.

These circumstances are supported by the Infrastructure API through the facility to associate an arbitrary number of Segment Maps with a model and to specify the connectivity between regions of the partitions which these maps represent. This high-level declaration of connectivity is translated by the Infrastructure API into the correct primitives appropriate to the particular target infrastructure that is in use.

2.3.3.   Addressing Model Entities

Interactions between the client scientific model and the Infrastructure API require a mechanism to mediate communication between them. BFG uses a "tag" mechanism for this purpose. The tag provides a link between the data involved in a particular operation (e.g. a coupling exchange) of a scientific model to the metadata describing that data. In a coupling operation, the metadata will include details of the target model of the coupling and also information about on which timesteps the coupling is actually to take place. This metadata

fulfils the same role as that provided by the information in the namcouple file in OASIS3. In BFG, the tag may be explicitly stated in an in-place operation (a put() or get()) or be inferred from the position of the data in an argument list (see IS-ENES deliverable D8.3 for more details).

The tag mechanism is an important part of the design of BFG supporting the isolation of science code from the implementation details of the infrastructure layer. The tag provides the link between the scientific model and the metadata related to the infrastructure operations invoked by the model.

In the current work extending BFG (beyond the current release, BFG2) to support parallel models, the tag mechanism is being generalised so that it can be used to refer to more general entities involved in infrastructure operations other than coupling exchanges, such as halo exchanges.

In the current BFG2 release of BFG, the tags are integers and the tag values have to be unique within a model. The infrastructure uses other context information, such as the (MPI) process id of the calling routine to ensure global uniqueness. In the current work developing BFG (beyond the current BFG2 release) to support parallel models, tags are now allowed to take a string value[1]. This is intended to provide a meaningful label for the developer that corresponds to the operation being invoked (e.g. a coupling exchange or a halo exchange). After specifying a tag name for a particular entity to the Infrastructure API, all future interactions between the model and the API which refer to this entity can be conducted in terms of the model's tag.

As in the current BFG2 implementation (see D8.3), tags are extracted from the BFG metadata describing models. A tag is associated with coupling data defined either explicitly in descriptions of any in-place calls (put and get) present in a model, or, for coupling data defined in argument lists, BFG allocates a tag based on the arguments position in the argument list.

Tags defined by a model form part of a global namespace that is managed by the Infrastructure API. The Infrastructure API assumes the role of resolving a tag from a string value to the entity to which it refers. Further details on the use of tags is provided in the following two implementation-oriented sections.

2.4.   Summary of Overview

This section has examined the general structure of the Infrastructure API and has considered the design decisions taken in arriving at this structure. The following section describes the facilities provided by the Infrastructure API from the perspective of a scientific model and illustrates how a model may employ the API.

3.   Implementation

This section provides an overview of the implementation of scientific models which make use of the Infrastructure API. Firstly, the role of model-specific metadata in conjunction with the BFG system is considered. Then, the API is examined from the perspective of a scientific model which employs its coupling and halo communications facilities. The implementation of the API in terms of MCT is discussed in the next section.

---

[1] An alternate representation is to encapsulate both the tag and the entity (e.g. data) it identifies within an object. This has the advantage of reducing the possibility of coding errors that arise by specifying an arbitrary tag.

3.1.　Code Generation from Metadata

This section describes the role of a coupling framework generator, such as the Bespoke Framework Generator (BFG), in translating model-specific metadata and multi-model composition information into data structures and executable code that are then employed by the Infrastructure API.

Note that, as explained in the introduction, a manual prototype implementation of the design described in this section has been produced in the current IS-ENES project, and this prototype targets MCT as the existing infrastructure. Implementing the code generation is planned for a future release of BFG.

A custom composite data type is generated by the BFG for each of the models in a coupled model and this data type contains model-specific details of whichever existing infrastructure (MCT, ESMF, OASIS) if being targetted. This type is called a ModelInfo structure. Each instance of this type contains variables which represent the state of one model in the coupled model. These variables consist of both book-keeping values, relating to the operation of the Infrastructure API itself, and values which are specific to the particular target infrastructure that is in use. In this respect, the ModelInfo type represents a repository of the attributes that characterise a particular model within the coupled model at run-time (i.e. within the coupled model's execution environment).

Figure 5 depicts the process by which model-specific metadata is translated into executable code which defines types, variables and procedures employed by the coupled scientific model during its execution.   The BFG system defines an XML-based metadata representation standard. As described in Section 4, BFG metadata describes models (including the coupling data they require and can provide), compositions and deployment. The metadata is parsed by the BFG and, in conjunction with a system of templates which assist in the production of syntactically-valid code, executable statements are generated which implement the coupling described by the input metadata.

A typical ModelInfo structure generated as a result of this process is illustrated in Figure 5 Many of the attributes within this structure are declared as pointers which are initialised by the modelsInit() procedure, as described in Section 9. The specific modelsInit()procedure is also generated by the BFG system and permits model-specific characteristics (such as the specific mechanism to be used to implement a coupling exchange) to be determined at runtime and stored in the Infrastructure API's state, in the appropriate ModelInfo strucuture(s), for example, for future reference in Infrastructure API calls made by the 'run' routines of models, for example, to initiate a coupling exchange of a particular field. This initialisation routine allocates storage for the values within each model's ModelInfo structure, populates values according to the names and connectivity specified by the BFG metadata, and maintains the internal state required by the Infrastructure API.

To better encapsulate the operations of the Infrastructure API, the ModelInfo type is not accessed directly by scientific models. Instead, a series of "lookup routines" are generated which allow the contents of a model's state, as represented by a ModelInfo structure, to be resolved from model code through the use of a "tag" identifier, as outlined in Section 13..

In order to isolate the code of the scientific model from changes in its underlying execution environment, the source code generated by the BFG is written to separate files which are included in scientific models by a mechanism appropriate to the language in which those models are written. The source file describing the ModelInfo type and its associated initialisation routine is therefore the primary mechanism by which per-model specifications defined in metadata are translated into characteristics of the coupled executable produced by

```
type ModelInfo
        character(len=32)                           :: modelName
        Integer                                 :: modelID
        Integer                                 :: tagCount
        character(len=32), dimension(:), pointer        :: tags, haloMap
        type(GlobalSegMap), dimension(:), pointer       :: gsmap
        integer, dimension(:), pointer          :: halodepth
        type(AttrVect), dimension(:), pointer          :: av
        type(Router), dimension(:), pointer            :: route
        type(Rearranger), dimension(:), pointer :: rearr
        Integer                                 :: comm
        integer, dimension(2)                   :: peer
end type ModelInfo
```

Figure 5: Illustration of a typical `ModelInfo` *structure for MCT*

the Infrastructure API.

In addition, the BFG generates an outline Makefile suitable for use with GNU Make. The makefile requires additional work from the developer to ensure it does the required job but the generated file includes sufficient detail to be a useful step towards compiling and building a coupled model. The makefile generator, along with other useful utilities, is available on the BFG portal described in Section 23

Changes to the metadata input to the BFG system are therefore immediately reflected in terms of the generation of an updated ModelInfo type on recompilation of the coupled executable. This technique, combining code generation from metadata with the use of an intermediate Infrastructure API, therefore provides substantial flexibility in terms of the ease with which it allows the transparent substitution of communications libraries and other changes in the execution environment, while preserving a uniform and consistent interface to the client model code.

3.2.   Scientific Model Implementation

This section considers the Infrastructure API from the perspective of a scientific model which wishes to make use of the facilities the API offers. As such, it begins by examining the structure of the execution environment which is shared by scientific models operating in conjunction with the Infrastructure API. It then describes the stages of execution through which a model will pass, and presents the API routines that are available to models at each of these stages.

3.2.1.   The Execution Environment

Code generated by the BFG from metadata, Infrastructure API code, existing infrastructure code libraries and model code are linked together in an 'execution framework'. This framework provides a global address space for entities relevant to the coupled scientific models it supports. This includes the models themselves, communications routines provided by the Infrastructure API, and the range of structures and types ancillary to the coupled models.

In the scheme employed by the Infrastructure API, the mapping of a mesh to the parallel threads of execution of a scientific model is implicitly determined by the particular decomposition of that mesh. The execution framework manages the creation of processes according to the characteristics defined by a mesh partition – this may be achieved through explicit programming or with the help of routines from the target infrastructure, if that provides appropriate support. Processes dedicated to the execution of one model's workload are created as a well-defined group by a top-level executable ultimately typically using the Message Passing Interface (MPI). This enables the Infrastructure API to exert a level of oversight and coordination over the processes participating in the execution of a scientific model and provides the mechanism by which per-model data and attributes are communicated by the Infrastructure API to the relevant model code (this functionality is ofter provided by a Driver layer in existing technologies).

As a result of this grouping, each executing instance of a scientific model has a Communicator type identifier available to it, through the local comm variable, that uniquely identifies the model of which it is a part and enables communication with its peers.

3.2.2. Scientific Model Initialisation

The Infrastructure API's execution framework (equivalent to the Driver layer) divides the lifespan of a scientific model into three distinct stages that are performed sequentially:

i.   Initialisation

ii.   Scientific calculation execution

iii.   Finalisation

These correspond conceptually to the construction, operation and destruction of an object instance in the object-oriented programming methodology.

The initialisation stage is the point at which the Infrastructure API establishes the data structures, inter-model routes and halo-exchange mappings that will subsequently be accessed in the calculation stage, based on metadata description expressed in an instance of the ModelInfo type. Memory allocation for these entities is managed by the API, with transitory allocations being freed on an ad-hoc basis during program execution, while globally persistent allocations are freed in the finalisation stage.

It is during the initialisation stage that code generated by the BFG system as a result of metadata specifications is executed. This generated code includes both static declarations and definitions of scientific model characteristics derived from its metadata, and dynamically allocated structures whose appropriate size and content can be determined only at runtime.

The initialisation stage can be further divided into global and subsequent per-model stages. In the global initialisation stage, the Infrastructure API establishes the runtime environment in which scientific models will execute, preparing the internal state and book-keeping that will underpin their subsequent execution. The initialisation of a particular model then creates the internal structures specific to the computational task that it will perform. One aspect of this initialisation is that each instance of a scientific model possesses a reference to its own metadata and execution state in the form of a 'this' pointer. This attribute of each model permits model code to perform introspective operations, querying its own state so as to establish an awareness of the execution environment which the Infrastructure API has constructed for it. This feature may be employed by the authors of scientific models to optimise code for particular environments, or to create model-specific operations which build upon the routines offered by the Infrastructure API.

3.2.3. Scientific Model Execution

Each scientific model coupled by the Infrastructure API is provided with one or more run

```
subroutine haloExchange(v, tag)

        real, dimension(:), target, allocatable, intent(INOUT)  :: v

        character(len=*)                                        :: tag

end subroutine haloExchange
```

Figure 6: Example of a haloExchange() subroutine definition provided in the Infrastructure API for use by a model to initiate a halo exchange

procedure within which the core of its scientific computation is performed. (The name of the run procedure is chosen by the developer, and a model may have several such procedures.) Control is passed from the Infrastructure API's execution framework to this procedure after the above initialisation phases are complete, and is resumed by the framework when this code returns.

It is during this stage of execution that a scientific model makes use of the runtime communications and data management facilities provided by the Infrastructure API. These facilities, described in Section 2, provide convenient and portable mechanisms to perform common tasks routinely encountered in scientific models. Notable amongst these are those primitives which facilitate communication between coupled models (coupling exchange) and also those that operate between the processes *within* a particular model (halo exchange).

Inter-model communication may be effected with the 'in-place' put() and get() operations. BFG2 can also generate inter-model communications for data passed into the scientific model through the argument list. These routines cause the communication of data to or from the address space of one process from or to that of another. Whereas the primitives provided by a basic communications layer may allow arbitrary data to communicated in a similar fashion, the put() and get() routines provided by the Infrastructure API consist of code generated from metadata specifically for the purposes of the models in question. As such, the scientific models linked with the Infrastructure API may make use of their own model-specific semantics when calling these routines, and are isolated from the details of the selected specific target infrastructure technology.

For example, an abstract representation of one version of the halo-exchange operation provided by the Infrastructure API's haloExchange() routine is depicted in Figure 6. Other variants of halo exchange, for example, supporting asynchronous exchanges, will also be provided.

This routine accepts a pointer to local storage (typically of a field) and a tag value identifying a particular entity to the Infrastructure API (i.e. in the global address space of the execution environment). On return, the pointer refers to a local copy of the field after the halo exchange, as specified in Section 2.

3.2.4.   Scientific Model Finalisation

This stage of model execution is conceptually the inverse of the initialisation stage. A

scientific model may have one, several or no finalise routines, depending on the particular behaviour of the model. If finalise routines exist, it is here that data structures and storage specific to the model are freed and returned to the execution environment. A model can call a single (Infrastructure API) finalise routine. Within this routine the Infrastructure API makes reference to its internal register of resources allocated for the model, both those originating from metadata and those from dynamic allocation during model execution, and ensures that those entities associated with the invoking scientific model are freed as appropriate.

As with the other routines provided by the Infrastructure API, this interface routine translates an operation that is general and broadly-specified at the level of the scientific model into a sequence of lower-level operations that are specific to the underlying, targetted, infrastructure technology in use. By doing so it allows the scientific model writer to concentrate on the scientifically meaningful portion of his code and to benefit from an abstract and simplified representation of coupling and communications operations.

3.3. Summary

The section has given a description of how the structures and programming principles that were presented in Section 2 have been translated into an Infrastructure API for coupled scientific models which provides these models with a convenient set of high-level types and operations that facilitate the implementation of scientific models.

One of the characteristics of scientific models employing this Infrastructure API is the isolation of the science models from the details of other models and details of supporting infrastructure; flexible, in terms of the ease with which the targetted existing infrastructure technologies may be substituted for each other; and portable, in terms of the consistent and uniform interface that is presented to scientific models irrespective of the particular environment in which they execute.

The following section considers aspects of the implementation of the Infrastructure API that are specific to the Model Coupling Toolkit (MCT), and describes the relation between the routines presented by the API to client scientific models and the underlying MCT operations to which they will be resolved.

4. Infrastructure API for the Model Coupling Toolkit (MCT)

4.1. Overview of the MCT

The Model Coupling Toolkit defines a relatively small series of types and operations which facilitate the organisation of data within models and the exchange of data in and between models in coupled models. This section describes how the primitives provided by MCT have been extended and combined by the Infrastructure API to make higher-level operations available to coupled scientific models.

In keeping with the mesh partitioning technique described in Section 7, MCT represents the entirety of a partitioned mesh as a GlobalSegmentMap. A GlobalSegmentMap is associated with the partition of a mesh across the set of processes associated with a model. In MCT, the GlobalSegmentMap is constructed from the bottom up; each process specifies (to the infrastructure) a set of segments that it owns. Each segment is defined using a "base + offset" notation that describes a contiguous region of sequentially numbered points in the global indexing used to describe the full mesh. Sparse partitions may therefore be represented by omitted regions of the mesh index space, and halo regions are indicated by the duplication of the haloed region of the mesh in the segments of multiple processes. MCT does not itself enforce a notion of ownership in definition of halo mesh points. This feature is

left to client scientific models to implement in a way that is meaningful within the context of their algorithms.

MCT describes communication connections between coupled scientific models using the concept of a route. Each route connects two models by describing a translation between the GlobalSegmentMaps defined by each individual model. The fundamental unit of transfer between coupled models is the AttributeVector. This is a composite type consisting of storage for the data to be transferred and an indexing system which permits each AttributeVector to contain multiple fields of both integer and real data. A particular set of data is specified using its "field name", a text label which uniquely identifies a field in the vector. The contents of AttributeVectors are transmitted and received across routes using the send() and recv() operations. The transfer of data values between the globally-indexed points owned by each processes of each model, as defined by the GlobalSegmentMaps, is managed transparently by MCT.

One characteristic of this scheme is that data used within a scientific model must be explicitly copied into an AttributeVector before it may be communicated to a peer process, and data received into an AttributeVector, as a result of a communication operation, must be explicitly copied into the local storage of a scientific model before that model may operate upon it. As will be described below, the Infrastructure API provides convenient routines which manage this copying in an efficient manner and allow scientific models to invoke abstract communications operations. These models are therefore not required to manage the low-level detail of communication between processes.

In terms of the architecture depicted in Figure 1, MCT provides a range of low-level coupling and communications routines that are encapsulated by the Infrastructure API. In this capacity as an implementation-level toolkit, MCT is interchangeable with OASIS and ESMF; the Infrastructure API maintains a consistent set of interface routines which remain constant despite changes to these lower levels of the execution environment. Scientific models which make use of the Infrastructure API are isolated from the details of these lower layers entirely.

4.2. Infrastructure API Extensions to the Model Coupling Toolkit

This section describes how some of the high-level operations provided by the Infrastructure API are implemented using MCT.

4.2.1. Halo Exchange

The Infrastructure API maintains an explicit representation of a haloed region, since this is a concept model developers require. This concept implies the notion of a set of local mesh points which are under the *ownership* of one of the processes participating in a model's execution. This is achieved by providing each process with two GlobalSegmentMaps, one for the local (but globally indexed) points 'owned' by the process and and one for the halo points required by the process. These two maps form the basis of the Infrastructure API's haloExchange() routine which permits the intra-model exchange of halo data between processes. Although MCT AttributeVectors underlie the communications (and coupling) operations, these are not exposed to scientific models using the Infrastructure API but instead are created and used transparently.

The haloExchange() procedure has the signature described in Figure 1818186. The tag parameter is first used to identify a particular MCT AttributeVector which corresponds the local storage of the calling process – essentially, the process' id and the tag are sufficient to identify this uniquely. Then, the AttributeVector representing the halo data for this process is similarly identified; both of these lookups are performed using several of the various name

resolution functions provided by the Infrastructure API to support the use of MCT; specifically these include the tagToModel(), tagToAV() and haloOf() functions. The AttributeVectors required by a model will have been set up and registered in the ModelInfo structure of the model during model initialisation, as described in Section 14.

The vector parameter to haloExchange() is a pointer to a vector within the address space of the calling process, whose contents are local to the calling scientific model. That is, this vector is part of the model's internal storage and operations performed upon it are local to a particular process. The contents of the vector are assumed to be the model's sequentially arranged local mesh points – the process will also maintain a mapping from the global index space of the mesh partition it owns to a local indexing scheme, as well as the mesh connectivity information required by the model.

On invocation of the haloExchange() routine, this data vector is used to initialise an MCT AttributeVector which is then communicated to any other processes for which any of these (global) points are halo points. In addition, a second AttributeVector is created which is populated with the calling processes halo points, the appropriate values being communicated by MCT from the processes which own these points. These communications operations are performed by MCT by way of its rearrange() procedure, and the construction of AttributeVectors appropriate as parameters to this procedure is performed by the importRAttr() routine.

The haloExchange() routine therefore modifies the vector pointer parameter passed to it so that, on return, it has been redefined to point to a vector, in a packed format, in which the calling processes sequentially arranged local points are followed immediately by its sequentially arranged halo points. That is, the caller's halo point values have been appended to the vector argument.

All of these interactions with MCT, and the allocation and freeing of the required storage for communication, are managed transparently by the Infrastructure API. The scientific model code therefore remains isolated from the detail of effecting these communication tasks.

4.2.2.　　Inter-Model Communication

In addition to communication within the group of processes executing a single scientific model, the Infrastructure API provides routines to facilitate the exchange of coupling data between models. The put() and get() routines accept similar parameters to the haloExchange() procedure in that they are passed a pointer to a vector local to the calling model, and a tag which allows the Infrastructure to access the metadata describing the coupling exchange. Within the put() and get() routines, the tag is used to identify first the model, and then the particular resource in a remote model which is being specified, by way of the Infrastructure API's name resolution functions described above. Each resource in a remote model corresponds to a particular route to that resource, and the correct route for communication to this model is selected. Finally, the MCT primitives Send() and Recv() are used to send from or receive to the appropriate MCT AttributeVector.

Routines are also provided to allow scientific models to interrogate the Infrastructure API's abstract representations of model entities and data types. These are transformed into the correct MCT statements to produce the equivalent information. For example, the localStorage() and haloStorage() functions return the size of the corresponding AttributeVector used by MCT to store the calling processes local and halo data respectively. The related commStorage() routine sets a pointer argument passed to the routine to point directly to the contents of an MCT AttributeVector, while preserving the encapsulation of this object. This enables a scientific model to manipulate data that is held by the underlying technology toolkit without having to copy this from the toolkit's storage into its own. This will be more efficient in circumstances where a model requires only a small number of accesses to a large data structure and copying this entire structure would result in many unnecessary

operations.

4.2.3.　　Runtime Symbol Resolution

Underlying the simplicity of the interfaces presented to scientific models by these Infrastructure API routines is the maintenance of an extensive internal state and a series of methods to update this state at runtime. The ModelInfo structure illustrated in Figure 5 is populated by way of several internal Infrastructure API routines which enable the structure's pointer attributes to act as generic containers for their types. MCT routes and GlobalSegmentMaps are added to this structure by way of the addRoute() and addMap() procedures. These procedures allocate storage for routes and GlobalSegmentMaps that is internal to the Infrastructure API. An indexing system which links tags, routes and GlobalSegmentMaps is maintained which permits objects to be retrieved from their containers using a "key", the tag, whose form is specified by the scientific model author; this technique replicates the general functionality provided by an associative array. This allows model authors to employ meaningful and convenient labels for the entities that exist in their model's namespace.

Although code generated by metadata of the sort that originates from the BFG system can specify the *types* that are required for a particular coupling, the MCT toolkit must be invoked at runtime to create instances of these types and populate these with appropriate values. This ability to modify model state at runtime enables substantial flexibility in establishing the coupling between models. Metadata may be used to comprehensively represent all of those scientific model characteristics that may be determined statically prior to execution, but it is frequently the case that other attributes of the execution environment cannot be determined until runtime. In this sense, the Infrastructure API's runtime abstraction layer builds upon and augments the coupling specifications provided by the BFG by providing a set of dynamic operations that work with the coupling framework's static code.

A circumstance in which the runtime characteristics of the Infrastructure API are particularly valuable is where the portability of a scientific model is an objective. The runtime management of model state enables it to adapt to the current execution environment in a manner that is not possible for metadata-generated code; as such, scientific model developers are freed from the burden of incorporating modifications into their models that seek to account for characteristics of a particular execution environment. Instead, ensuring that the operations of the scientific model are translated into the correct primitives to correspond to a particular local environment becomes the responsibility of the Infrastructure API.

5.　　Infrastructure API – notes on OASIS3-MCT and ESMF

In order to assess how the Infrastructure API would target OASIS3-MCT as an implementation technology, the two model tutorial provided with the OASIS3-MCT download was simplified to a case equivalent to that implemented using MCT. The two models were modified to use the same, simple, unstructured mesh with each model partitioning the mesh differently, using the ORANGE (segmented) partitioning type in OASIS. The association of the mesh with geographical information was also removed, to simplify the problem and the namcouple file modified to reflect these changes.

The steps required to couple the models with OASIS3-MCT and the implications for using OASIS3-MCT as a target for the Infrastructure API are as follows. In the model initialisation phase:

1. Set up the partitioned mesh on a per-process basis for each model. This requires the Infrastructure API to call the oasis_def_partition() operation with the appropriate

ORANGE partition information, which the Infrastructure can compute from the mesh partition definition it has.

2. Set up the variable definitions. This is equivalent to setting up the AttributeVectors used in MCT (and in OASIS3-MCT, the oasis calls themselves translate this information into MCT calls). This involved defining the shape of the data, allocating an id and name for the field, and passing the dimension information and partition information into the oasis_def_var() operation. The required information can be obtained from the Infrastructure ModelInfo structure set up during the coupled model initialisation phase.

3. End the definition phase with a call to oasis_enddef().

In the models initialise, run and finalise sections, the Infrastructure API calls to put() and get() (with appropriate tags) simply have to be translated into oasis_put() and oasis_get() calls for the appropriate fields. The time information required in these calls can be set from the (BFG) metadata describing the coupling, which is available to the Infrastructure API. BFG would also generate the appropriate namcouple file, which would be read during the OASIS initialisation invoked during the Infrastructure API's initialisation phase.

This exercise suggests that there are no issues with targeting OASIS3-MCT from the Infrastructure API.

A similar exercise has been undertaken to explore targeting ESMF infrastructure (not superstructure) for halo exchange (OASIS does not provide support for halo exchange). The targeting of ESMF for coupling is still a work in progress at this point. There is already a level of support for ESMF as a target in the existing BFG2 (see the previous report, D8.3, "Towards Flexible Construction of ESMs using BFG".

6.   Update on BFG2 Online Access

In the previous report, D8.3, "Towards Flexible Construction of ESMs using BFG", work on a BFG portal was discussed. The portal can be accessed at: http://bfg.cs.man.ac.uk. In D8.3, four lines of possible future work were mentioned:

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).

Tasks 2 (Makefile generator) and 3 (BFG1 to BFG2 translation) have been completed and these tools are now available through the BFG portal. Some basic functionality for Task 1 (editing) and task 4 (visualisation) have been added to the portal and these are planned to be further developed in the future.

The BFG2 Makefile generator, takes BFG2 coupled XML documents and attempts to create an appropriate Makefile to compile the generated code. As BFG2 coupled XML documents do not keep any information about compilation, a separate configuration file is read to specify appropriate compilers, linkers and libraries. The BFG1 to BFG2 translator automatically translates BFG1 coupled documents into BFG2 coupled documents. As the BFG1 API is a subset of the BFG2 API, any BFG1 models can be used in BFG2 couplings.

7. Update on other work including support for Climate and Integrated Assessment Modelling

Deliverable 8.3 mentioned a number of areas that we were either working on, or planning to work on, for our remaining effort in WP8 task 3. This section summarises progress on these aspects.

1. Import and Export of models. There is now prototype support for exporting BFG2 models to either OASIS3-MCT ot ESMF. Model export means that models written to BFG's API can be automatically wrapped to make then compliant to other coupling systems. Therefore a BFG model can be used as either a native OASIS-MCT model or a native ESMF component with no change to the source code, thereby providing one step towards mujlti-framework support. This will be made available in the next BFG2 release. However, staff changes have meant that the importing of models has not yet been addressed.

2. Support for Parallel Models. Good progress has been made here, as discussed in the main part of this document, and we have working manually written solutions for OASIS3-MCT, MCT and ESMF which will serve as reference implementations to extend the BFG2 metadata appropriately and ultimately allow us to add code generation support in BFG2.

3. Open source BFG. The latest releases of BFG2 are now being made available as source code on the BFG web site (currently, version 1.1.2, see: http://www.cs.man.ac.uk/cnc/projects/bfg). There are also plans to host the repository on CCPForge (*ccpforge.cse.rl.ac.uk)* with open read access to the repository, but that has not yet been completed.

4. Updated BFG2 implementation. BFG2 has been further updated to reduce the reliance on XSLT which has been replaced with Python XML. This has helped with code maintenance, allowing Python language support to be added more easily and improved the speed of code generation significantly.

5. Validation of XML input. Little progress has been made with the checking of consistency between metadata and code, however a compresensive XML input check routine has been added to the BFG2 toolset. This check is performed before all BFG2 transformation tools by default and picks up many of the errors that users can make in the input XML. It may also be run on its own and may be used from the portal in this way. Whenever a new error is reported by a users, that is not picked up by the tool, a check for this error is subsequently added in the next release.

6. New Examples. A UM model example was going to be tested using BFG's program compliance. This has not been completed due to staff changes (the person tasked to do this moved jobs part way through the project).

8. 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.

In this report, work has been presented which is designed to extend BFG to support parallel models using the emerging non-uniform meshes, designed to overcome the known scalability limitations of traditional long-lat meshes used in global models. A prototype design and development of a metadata-driven, run-time configured, Infrastructure API has been presented. The API is consistent with the goals of BFG to isolate science code from infrastructure technology details. A specific targetting of the API to the existing Model Coupling Toolkit has been presented and exploratory work has been undertaken for targetting OASIS3-MCT and ESMF.

It is anticipated that this work will inform the current debates in the ESM community concerning framework interoperability and the possible adoption of a single, common software infrastructure as discussed in the IS-ENES foresight report, "Infrastructure strategy for the European Earth System Modelling community 2012-2022",(https://is.enes.org/the-project/communication/ENES foresight.pdf) and in the recent report, "A National Strategy for Advancing Climate Modeling", from the National Academy of Sciences in the U.S.A, 2012 (https://download.nap.edu/catalog.php?record_id=13430).