HTW des Saarlandes
Wintersemester 2012-2013
Projektarbeit

Domine Specific Language

Dozent: Prof. Dr. Ralf Denzer

Teilnehmer:

Manhal Anwar 3479129

Gilles Baatz 3536491

Sergej Herzog 3502031

Bo Lin 3565556

Daniel Meiers 3538990

Jochen Palz 3479374

David Rupp 3502236

Oliver Wolf 3502007

31. March 2013

Contents

List of Figures

1 abstract

2 introduction

2.1 Background

Modern information and communication technologies are widely used in almost every field. It has also influenced the environmental science.

2.2 Project Goals

A more universal approach to describe environmental models is needed. A domain specific language was proposed as a solution to this need. A plain DSL will not be enough to solve all problems involved in describing an environmental model in such ways that it can be fully described and exchanged without additional information. The following features were defined as necessary for a language/system that allows to fully describe and exchange models:

- Domain-specific data structures
- Rich model interfaces
- DSL handling typical operations
- Support for different modeling paradigms and frameworks
- Account for modeling uncertainty
- Model transparency and defensibility of results

After the first step of information gathering the group decided to organize the features by priorisation and practicability, to decided which goals are achievable in the given time.
The following order was defined:
priority 1

- Domain-specific data structures
- DSL definition with typical operations
- Account for modeling uncertainty

priority 2

- Rich model interfaces
- Model transparency and defensibility of results

priority 3

- Support for different modeling paradigms and frameworks

As a result of the lack of practical modeling experience the group decided that an universal approach of these feature is not practical. A more practical approach was decided. To gather experience a concrete model will be implemented with different frameworks and then a DSL will be defined with the necessary expressions to implement this model including the ontologies for the data used in the model. With the time given the other features mentioned in the priority list above will only be worked on in a conceptual way. A detailed description of the project proceeding is shown in the paragraph.

2.3 Project Proceeding

2.3.1 Step 1 - Information Phase

In this phase the project participants have to do literature research on environmental modeling in general, and the state of the art of modeling environments in

special. Several modeling environments are under closer examination to get an idea of how the modeling process looks like and what features will be needed in a DSL.

2.3.2 Step 2 - Designing an DSL

With the gathered information from step 1, a Domain specific language (or something alternative) shall be designed that enables modelers to define their models in an intuitive way. This step shall be based on sample model (not to complex, not to simple). And just for one modeling paradigm. The decision for a concrete example must be chosen from a wide used modeling paradigm.

- *Step 2.1 - Ontology based data types*
  The first step is to find out if there are already ontologies witch define data types that can be used in models. If there are, they must be analysed for their value for this project. Next a mechanism is needed to dynamically import data types, characterized by ontologies, to parametrize the DSL.
- *Step 2.2 - DSL development*
  With the experience gathered in the former steps a DSL must be defined with the necessary capability to implement the example model.The modeling uncertainty problem must be considered when implementing the DSL and the data types.
- *Step 2.3 Concrete Implementation*
  Some kind of toolchain will be necessary to implement the concrete model with the defined DSL and data types definitions.

2.3.3 Step 3 - Conceptual work

After the implementation conceptual work will be done on the features with priority 2 and 3.

3 Vision of DSL

This chapter describes the, above mentioned, features for a modeling environment. They were initially proposed by Athanasiadis [Ath12], and are further explained in the following sections.

3.1 Domain-specific data structures

Domain-specific data structures can be used by the modeler to semantically describe the following elements in the DSL.

- units and quantities
- accuracy
- spatial and temporal scales and extents
- quality and provenance information of data sources and results

The newly created models will consist of independent logical models and their observations. This is a novel approach, as it is a semantical representation of environmental data sets. In other words, the code for one entity is logically capsulated and separated from other entities.

The main idea to achieve this, is to connect the DSL with specific environmental modelling ontologies, as these define the semantics of the environmental terminology.

3.2 Rich model interfaces

Rich model interfaces should allow the modeler to share their models in scientific workflows. Therefore the models must be enriched with the following metadata in machine-readable formats.

- incorporating model assumptions
- pre- and post- conditions
- prerequisites for reuse

3.3 DSL handling typical operations

To simplify the modelers work the DSL should be able to automate typical operations, among this:

- scaling
- averaging
- interpolation
- unit conversions

The language will be able to treat appropriately intensive and extensive quantities, for example by calculating the weighted mean when joining two intensive quantities.

3.4 Support for different modeling paradigms and frameworks

Athanasiadis proposed that the DSL should be paradigm agnostic and should also be able to be compatible to several frameworks. Nevertheless this point makes the development of the DSL very complex. Therefore a focus was set on the System Dynamics paradigm.

3.5 Account for modeling uncertainty

The model environment should be able to compute the modeling uncertainty and quality information. This should be achieved with confidence intervals, which solve different sources of uncertainty like:

- random sampling error and biases
- noisy or missing data
- approximation techniques for equation

An example would be the standard error propagation. Two variables x and y are given, their mean and variance are represented by the following tuples (µ x , ?x) and (µ y , ?y). If their difference is calculated and saved in z, the mean and variance of z is automatically represented in the tuple (µ x - µ y , ?x + ?y). Despite the importance of this feature, it was out of the scope of this project and no further investigations were made.

3.6 Model transparency and defensibility of results

One feature of the environment should be model transparency and defensibility to explain the results of the model. This means that for each model output a history of operations on primal sources, enabled by the enrichment of metadata, must exist.

4 Research Phase

4.1 Current Modeling Environments

4.1.1 Simile (Simile Visual Modelling Environment)

Simile is a visual modelling environment, developed originally for the dynamic modelling of ecological and environmental systems, which supports a wide range of ways of representing space. Therefore, it addresses both of the above issues in a single environment.

Visual modeling environment

- Build model without learning programming
- Display concept & relationship using diagram, for example (stocks, flows and influences)
- GUI for simulation control & specifying inputs
- Visualize & compare the model behaviors in graphs & tables

What can we use it for?

- Rate expressions for biological, chemical, or physical processes (differential equations), for example: Mass balance or accounting, Growth dynamics, Respiration
- Scenario analysis, for example: Effect of flow velocity on gas exchange between river-atmosphere

Technical implementation:

Architecture

The following diagram 4.1 shows the main components of Simile, and how they relate to each other.

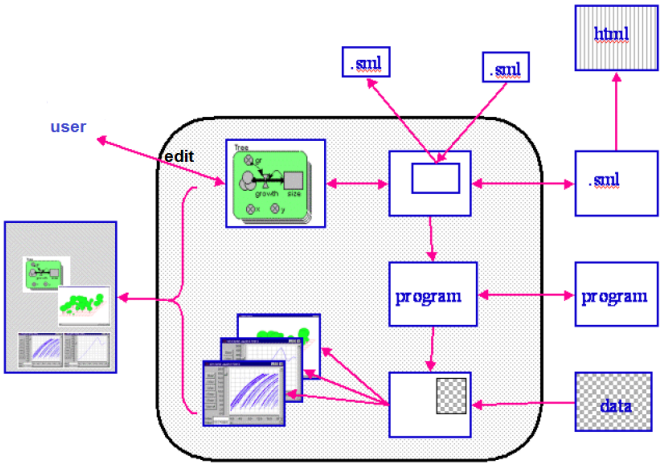

Figure 4.1: Simile Architecture

In the middle-left, the user interacts with Simile through a graphical user interface (GUI) to edit the model diagram and equations. In the process, Simile builds up an internal representation of the model, which may contain submodels. The model or a submodel (submodel will be explained later) can be saved to file (.sml extension) (arrows going to the top and to the right). Also, a model or a submodel can be loaded from file (arrows coming in from top and right).

To run a model, the internal representation is used to generate a program in tcl (Tool command language) or C. (In fact, not shown here is that in C Simile first generates the source code, then compiles it as a DLL (Data Definition Language)). This program can also be saved to file, it is possible to run a previously-built model as a stand-alone executable without having access to the Simile environment. The program may then need to be combined with external data sets, containing parameter and tabulated values. After run, Simile calls on various display tools to show the results of the simulation. The displayed results of the simulation may then be exported, along with the model diagram, to produce a postscript file that can be used to produce a handout or poster about the modelling exercise.

At the top-right of the diagram, you will notice that the saved model can be

used to generate an html description of the model. This is handled by a program (written in Prolog) that is totally independent of Simile itself. The html generator is the first of many such tools, and ultimately we envisage that many groups around the world will be writing their own for processing Simile models in new and useful ways.

Properties of the modeling:
Simile has a number of features

Visual modelling:

Simile supports a two-phase approach to model construction. The first involves the drawing of diagrams that show the main features of the model and the second involves fleshing-out the model-diagram elements with quantitative information on the relevant values and equations.

System Dynamics:

Simile allows models to be formulated in System Dynamics terms, as compartments (stocks, levels) whose values are governed by flows in and out. This can be considered as a visual language for representing differential-equation models, with a compartment representing a state variable, and the rate-of-change being the net sum of inflows minus outflows.

Disaggregation:

Simile allows the modeller to express many forms of disaggregation: e.g. age/size/sex/species classes. This is achieved by defining how one class behaves, then specifying that there are many such classes.

Object-based modelling:

Simile allows a population of objects to be modelled. As with disaggregation, model designers state how one member behaves, then specify that there are many such members. In this case, the designer can add in symbols denoting the rules for creating new members of the population, and for killing off existing members. Individual members of the population can interact with others.

Spatial modelling:

It follows that spatial modelling in the system is simply a special form of disaggregation. One spatial unit (grid square, hexagon, polygon, etc.) is modelled, then many such units are specified. Each can be given spatial attributes, such as area or location, and the proximity of one unit to another can be represented.

Modular modelling:

Simile allows any model to be inserted as a submodel into another. Having done this, the modeller can then manually make the links between variables in the two components (in the case where the submodel was not designed to plug into the main model); or links can be made automatically, giving a (plug-and-play) capability. Conversely, any submodel can be extracted and run as a stand-alone model, greatly facilitating testing of the submodels of a complex model.

Efficient computation:

Models can be run as compiled programs. In many cases, these will run as fast as a hand-coded program, enabling Simile to cope with complex models (100s equations; 1000s object instances). While larger or institutional spatial databases are likely to contain millions of object instances, the complexity of modelling, rather than the efficiency of computation, means that dynamic spatial modelling tasks are often more modest in size.

Customisable output displays and input tools:

Simile users can design and implement their own input/output procedures independently. In particular, they can develop displays for model output that are specific to disciplinary norms or other requirements. Once developed, these can be shared with others in the relevant research community.

Declarative representation of model structure:

A Simile model is saved in an open format as a text file (in Prolog syntax). This means that others can develop tools for processing Simile models in novel ways. For example, one group may develop a new way of reporting on model structure, while another may wish to undertake automatic comparison of the structure of two similar models. It also opens the way for the efficient transmission of models across the Internet (as XML files), and for the sharing of models between different modelling environments.

In fact, Simile has no particular spatial representations built into it, rather these are specified by the user in Simile's modelling language and this means that modellers have considerable flexibility in just how space is represented. They are not restricted to some pre-defined spatial framework. One model can include both field and object views, polygonal, rectangular and hexagonal areal units, 3D units (e.g. cubes), and point and linear features, all referenced to a common co-ordinate system. Together with appropriate visualisation tools, this flexibility enables a very wide range of dynamic spatial models to be developed.

The Simile visual modelling environment allow to:

- draw the elements of model, and the relationships between them
- add influences between related variables
- using simple mouse actions, to re-arrange the elements, annotate the diagram or add graphics

System Dynamics symbols

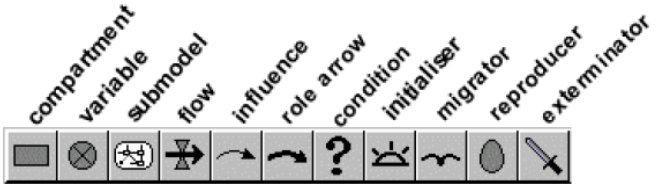

Figure 4.2: Simile Symbols

- *compartment :*
  The compartment symbol is used to represent a quantitative state variable
- *Variable :*
  A variable is used to hold one or more values. The value or values come from a mathematical expression
- *Submodel :*
  In Simile, a submodel is a round-cornered box that encloses a number of model-design symbols, including possibly other submodels
- *Flow arrow:*
  The flow arrow is used to specify a term contributing to the rate of change of a compartment
- *Influence arrow :*
  To say that (A influences B) means that A is used to calculate a value for B
- *Role arrow :*
  Role arrows are usually used in pairs, with each arrow going from a submodel to a submodel. The combination of submodels and role arrows is used to denote an association between the objects represented by the submodels at the start of each role arrow
- *Condition :*
  A condition element is placed inside a submodel to indicate that the existence of the submodel depends on some condition

- *Initial-number :*
  The initial-number element is used to specify the initial number of members in a population of objects

- *Migrator :*
  The migrator element is used to specify the rate at which new members of a population are created

- *Reproducer :*
  The reproducer element is used to specify the rate at which each member of a population creates new members

- *Exterminator :*
  The mortality element is used to specify the rule for killing off a member of a population of objects

In common with other visual modelling environments, Simile models are built in two phases. First, the modeller produces a diagram. Then, the modeller quantifies the model by entering numeric values and equations for the various components of the model.
The diagramming icons are chosen from the Toolbar of simile, the compartment, flow, variable and influence, are concerned with conventional System Dynamics modelling, shown in Figure 4.3.

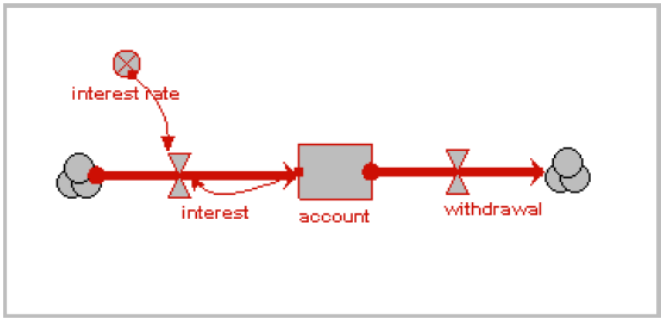

Figure 4.3: the model diagram

System Dynamics (SD) is a common dynamic modelling paradigm within the ecological and environmental research communities. A SD model consists of compartments (stocks, levels, storages) connected by flows, with subsidiary variables

for representing parameters and intermediate variables, and influence arrows to show which compartments and variables are used in the calculation of flows and other variables. Essentially, SD is a cosmetic language for defining differential- or difference-equation models: differential equations if the equations are taken to define continuous change; difference equations if the time step is taken to be unity.

Concern following problems of simile

- effort and skill needed to program the models
- the lack of transparency in models implemented as programs
- and the lack of reuseability of models and submodels

4.1.2 esmf (Earth System Modeling Framework)

The ESMF is an abbreviation for Earth System Modeling Framework. ESMF is an Open Source project used at a broad spectrum of research and operational centers, including the National Weather Service, United States Army Air Forces, the Navy, NASA, and universities.
ESMF is used for building and coupling weather, climate, and other Earth science models. This is high-performance software that facilitates interoperability, performance portability, and reuse in numerical weather prediction, climate, data assimilation, and other Earth and space science applications. These applications are computationally intensive and usually run on supercomputers. ESMF includes re-gridding tools for composing complex, coupled modeling systems, and data structures and utilities for developing individual models. The architecture for that is defined by the ESMF team.

Technical implementation:
ESMF provides a complete set of Fortran interfaces and some C and C++ interfaces.
ESMF is based on principles of component-based software engineering. It is a reuse-based approach to defining, implementing and composing loosely coupled independent components into systems. In ESMF, components can create and

drive other components so that an ocean biogeochemistry component can be part of a larger ocean component.

Architecture
The view of the framework consists of two layers, an infrastructure of utilities and data structures for creating model components, and a superstructure for coupling them. User code is placed between these two layers, to make calls to the infrastructure libraries and be scheduled and synchronized by the superstructure. A hierarchical combination of superstructure, user code components, and infrastructure are joined together to form an ESMF application. The architecture that resembles a sandwich is shown in Figure 4.4.

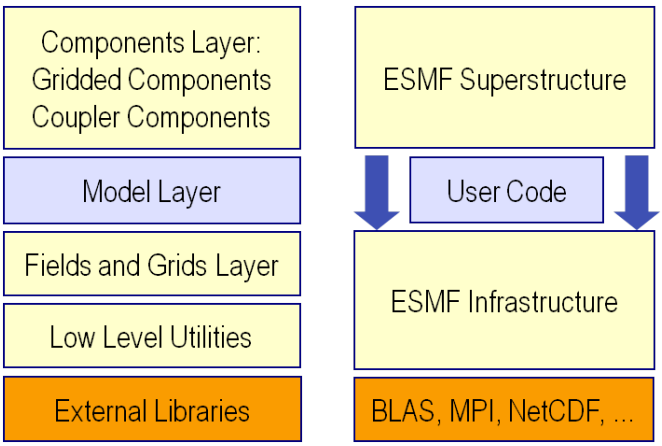

Figure 4.4: Schematic of the ESMF (sandwich) architecture

The ESMF architecture is scalable, flexible paradigm for building highly complex climate, weather, and related applications from components such as land models, atmospheric models, and data assimilation systems. The ESMF is a way of developing components which can be used in many different user written applications. Model components that adopt ESMF can be used in different contexts with minimal code modification. Furthermore they may be incorporated into other ESMF-based modeling systems within the Earth science community. The components can create and drive other components. In order to take the outputs from one component and transform them into the inputs that are needed to run another component, couplers is used. In ESMF, couplers are also software components.

Superstructure layer
The ESMF superstructure is a unifying context. User components are interconnected with this context. The superstructure is the means by which models are converted into components and coupled.
Classes called Gridded Components, Coupler Components, and States are used within the superstructure to achieve this flexibility. Gridded Components are major Earth system model components such as land surface models, ocean models, atmospheric models and sea ice models. Components used for linear algebra manipulations for example in state estimation are also created as Gridded Components.

The Figure 4.5 shows a typical building block for an ESMF application consists of a parent Gridded Component, several child Gridded Components, and a Coupler Component. The parent Gridded Component is called by an application driver. All ESMF Components have initialize, run, and finalize methods. The diagram shows that when the application driver calls initialize on a parent Gridded Component, the call cascades down to all of its children. The result is that the entire (tree) of Components is initialized. The run and finalize methods work on the same principle. In this example a hurricane simulation is built from atmosphere and ocean Gridded Components. An atmosphere-ocean Coupler Component handles the data exchange between the ocean and atmosphere.

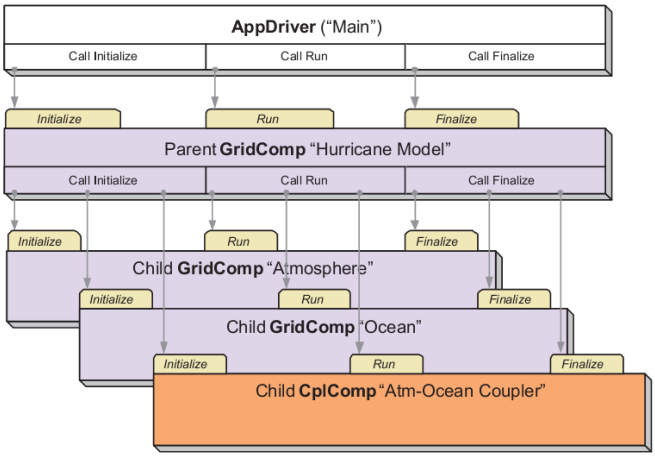

Figure 4.5: Building block for an ESMF application

User code components under ESMF use special interface objects for Component to Component data exchanges. These objects are of type import State and export State. The methods of these types allows user code components to do things like fill an export State object with data to be shared with other components or query an import State object to determine its contents.
The import State and export State abstractions are designed to be flexible enough so that ESMF does not need to mandate a single format for fields. For example, ESMF does not prescribe the units of quantities exported or imported. However, ESMF does provide mechanisms to describe units, memory layout, and grid coordinates. This allows the ESMF software to support a range of different policies for physical fields.
The Gridded Component class describes a user component that takes in one import State and produces one export State, shown in Figure 4.6.

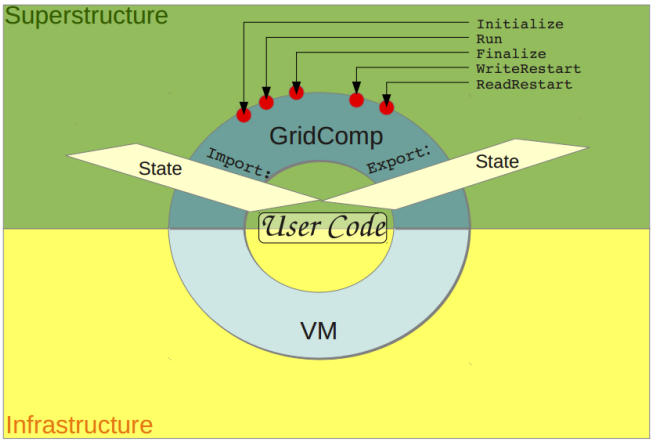

Figure 4.6: Architecture-Components and States

Infrastructure layer
Infrastructure is a standard software platform for enabling interoperability (developing couplers, ensuring performance portability). The infrastructure layer represents support for Physical Grids, Regridding, Decomposition/composition, Communication, Calendar and Time, I/O, Logging and Profiling. The infrastructure represents data structures and tools (Utilities) that modelers can use within their own code.

Data structures
Model data is contained in a hierarchy of data structures. The user can reference a Fortran array to an ESMF Array or Field, or retrieve a Fortran array out of an ESMF Array or Field.

- Array: holds a Fortran array (with other info, such as halo size)
- Field: holds an ESMF Array, an associated Grid, and metadata
- Bundle: collection of Fields on the same Grid bundled together for convenience and data locality. Bundles are intended for performance optimization, by sharing collective communication, IO, and regridding

Utilities

- Time Manager Utilities include Calendar, Clock (needed for superstructure), Time, Time interval, Alarm. These can be used independently of other ESMF utilities.
- Configuration Attributes (replaces namelists)
- Message logging (methods for writing error, warning & informational messages)

Integration of user model in ESMF
The steps for converting a user model into an ESMF component are mentioned here:

1. Separate code into a new module with initialize, run, and finalize stages
2. Create register routine
    - Register the initialize, run, and finalize routines
    - Make the register routine public
3. Include framework module

Hello World App

Hello World App in Fortran. For ESMF, Hello World demonstrates a single component driven from one layer above using SetServices() and Gridded Components in practice, shown in Figure 4.7.

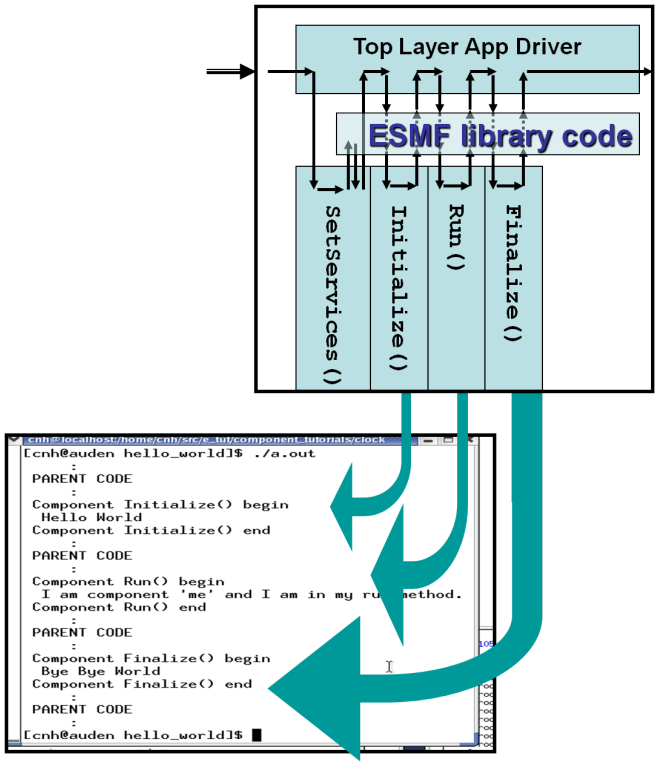

Figure 4.7: Hello World App in Fortran

Hello World in Fortran has two source files:

- MyWorldGridCompMod.F90 contains the SetServices(), Initialize(), Run() and Finalize() code.
- hello_world_app.F90 contains the top layer app driver.

Properties of the modeling

In ESMF, user data are represented in the form of data objects such as grids, fields, and arrays. The user data within a component may be copied or referenced into these ESMF objects. ESMF can associate metadata with data objects. The metadata, in the form of name and value pairs, is grouped into packages.

These metadata packages can be written out in XML and other standard formats.

Object-based modeling:

The ESMF Application Programming Interface (API) is based on the object-oriented programming concept of a class.

Flexible data and control flow:

Import States, export States, Gridded Components and Coupler Components can be arrayed flexibly within a superstructure layer. Using these constructs, it is possible to configure a set of concurrently executing Gridded Components joined through a single Coupler Component of the style shown in Figure 4.8. It is also possible to configure a set of Components with multiple point to point Coupler Components shown in Figure 4.9. The set of superstructure abstractions allows flexible data flow and control between components. However, components will often use different discrete grids, and time-stepping components may march forward with different time intervals. The ESMF infrastructure layer provides elements to manage this complexity.

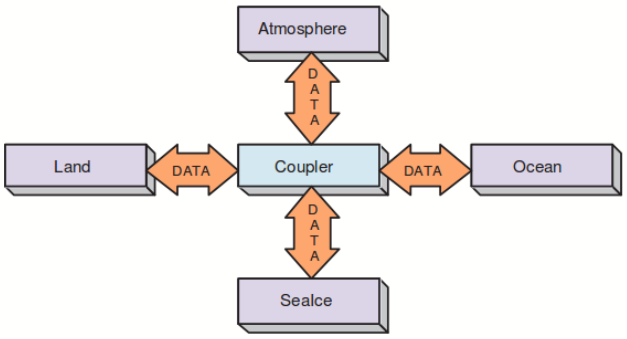

Figure 4.8: ESMF supports configurations with a single central Coupler Component

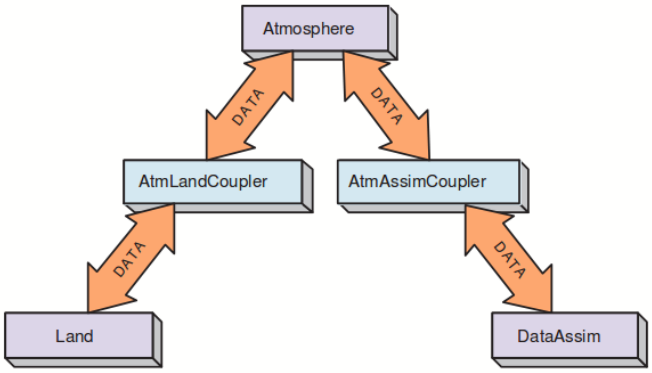

Figure 4.9: ESMF configurations with multiple point to point Coupler Components

Parallel computing:

ESMF runs on most high performance parallel computing platforms, including IBM, many Linux variants, Cray, Compaq, more. It supports Message Passing Interface (MPI), Open Multi-Processing (OpenMP) and hybrid user codes.

Sequential execution:

ESMF supports sequential mode (Single Program Multiple Datastream) but only very limited concurrent mode (Multiple Program Multiple Datastream) of execution.

System Dynamics:

The major classes in the ESMF superstructure are Components, which typically represent large pieces of functionality such as models, model couplers, dynamics and physics packages, and States, which are the data structures used to communicate data between Components.

Visual modeling:

ESMF provides regridding, also called remapping or interpolation. Regridding may be needed when communicating data between Earth system model components such as land and atmosphere, or between different data sets to support operations such as visualization.

Modular modeling:

ESMF is a software package for building modular, high performance modeling and data assimilation applications.

Pros and Cons
ESMF provides a toolkit that components use to increase interoperability, improve performance portability and reuse common utility code. Componentization is simplified by dividing user models into initialize, finalize, and run routines. The process is scalable, because additional components are created the same way.
The framework provides a set of portable, robust, performance optimized libraries for data transfers, time management, and other common modeling functions. In order to take advantage of the ESMF infrastructure users should extensively rewrite their codes. Or users may decide to wrap user-written components in ESMF interfaces in order to adopt the ESMF architecture and use framework coupling services.

Single execution mode:

It is not expected that a single Gridded Component be able to function in both sequential and concurrent modes, although Gridded Components of different types can be nested. For example, a concurrently called Gridded Component can contain several nested sequential Gridded Components.

No Transforms:

Components must exchange data through ESMF-State objects. The input data are available at the time the user Component code is called, and data to be returned to another Component are available when that code returns.

In a coupled ocean-atmosphere model, for example, the task of replacing one ocean model with another model from a different organization often requires a major redevelopment effort. One of the main goals of the ESMF is to develop a standard interface which will clearly separate model component and couplers, so that interoperable components can be shared and reused.

What is OMS good for?
OMS supports component based software engineering. Models and components are regarded as objects in OMS. The OMS framework introduces a new style of component modelling. No interface should be implemented. OMS uses Meta data with annotations to specify the input and output. This new kind of annotations makes it easier for modelers to identify the interactions of different components. The modelers will have a better overview of each component. Data flow through the components will be clearer for them. The modelers can profit from this feature to create a large component combining different subcomponents to fulfill a complicated task.
The OMS framework provides simulations features for modelers to test the model created by them. Simulations are model applications. It consists of model, input data, output management, execution type, execution environment. OMS provides a DSL to run the simulations. DSL is a mini-language that represents constructs of a given domain. With a DSL, simulations can be created from different tools, for example, IDEs, OMS console. With the simulation feature, the modelers are able to evaluate and improve their models. OMS provides diagram visualization of the simulation results. Modelers can use this feature to get a deeper insight of the functions of their models.
OMS is designed to support delivery of science relating to agricultural and natural resource management. It is so versatile usable that it can inter-operate with other frameworks of agricultural modeling worldwide.

4.2 oms

What is the disadvantages of OMS?

- Not stable. Many exceptions
- Little help documentations
- Costs much time to make OMS work

4.3 problems

4.4 declarative modelling

4.5 semantic approaches

5 toolchain

6 conclusion

7 glossar

Bibliography

[Ath12] Ioannis N. Athanasiadis. *Short note on Domain Speci?c Languages for environmental modelling*. 2012.