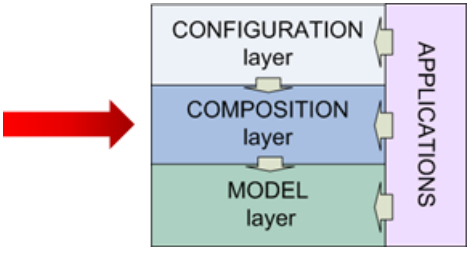
European Commission

BioMA

Composition Layer



Reference documentation

Joint Research Centre

Release	Issue	Date
1	1	May 2013

Disclaimer

On any of the MARS pages you may find reference to a certain software package, a particular contractor, or group of contractors, the use of one or another sensor product, etc. In all cases, unless specifically stated, this does not indicate any preference of the Commission for that particular product, party or parties. When relevant, we include links to pages that give you more information about the references.

Feel free to contact us, in case you need additional explanations or information.

Contents

About this document

1

This document is targeted to the advanced users of the BioMA Software Framework who want to deepen their knowledge of the Composition Layer of BioMA, in order to compose new modelling solutions and/or develop components.

The topics are organized as follows:

Topic	Description
"About the Composition layer" on page 5	• What the Composition layer is with respect to the BioMA framework architecture • Main purposes of the Composition layer
"Language reference" on page 9	• Classes description • ModelRunner execution cycle • Pre-built components • Using the CLIC application

Web resources of interest

Resource	What you will find
BioMA Framework User Guide	A comprehensive Web-based help that provides a description of the BioMA framework, as well as an overview of its main components.
BioMA Framework Portal	The BioMA portal, which links to all components' available documentation.
CropSyst Modelling Solution Documentation Wofost Modelling Solution Documentation	Document aimed to the BioMA framework end users, which describes the CropSyst and Wofost modelling solutions, respectively.

About the Composition layer

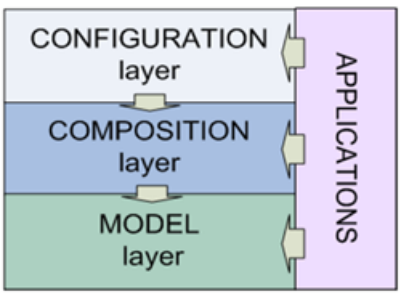
2

The BioMA Framework architecture

The composition layer is one of the layers of the BioMA framework architecture.

Tip:

For further information of the architecture and the components of the BioMA framework, please refer to the Web-based User Guide.



The **composition layer** relies on the **model layer**. This latter provides a set of strategies (containing the model logics), which are organized in components.

Usually, a component is coded as a library (.dll) containing all the strategies that relate to the same simulated physical environment. For example, the model layer provides a component containing strategies to simulate the plants growth, a component to simulate the soil hydraulic properties, a component to simulate weather variables, and so forth.

The aim of the composition layer is to compose several simulation components into one '**modelling solution**'.

The modeling solution represents a complex system made by many interacting environments.

Each simulation component encapsulates the logics of a well-defined part of the model and/or the logics to retrieve the required data to run the model (e.g. the weather data). Components can be developed independently and they are not directly inter-related.

Purposes of the Composition layer

The main purposes of the composition layer are the following:

- To define reusable components
- To compose several components into a unique composition (modelling solution) in order to simulate a more complex system
- To provide the applications with aggregated information (e.g., modelling solution's inputs/outputs/parameters/metadata)
- To define a cycle of simulation
- To collect the components data into an unique model output

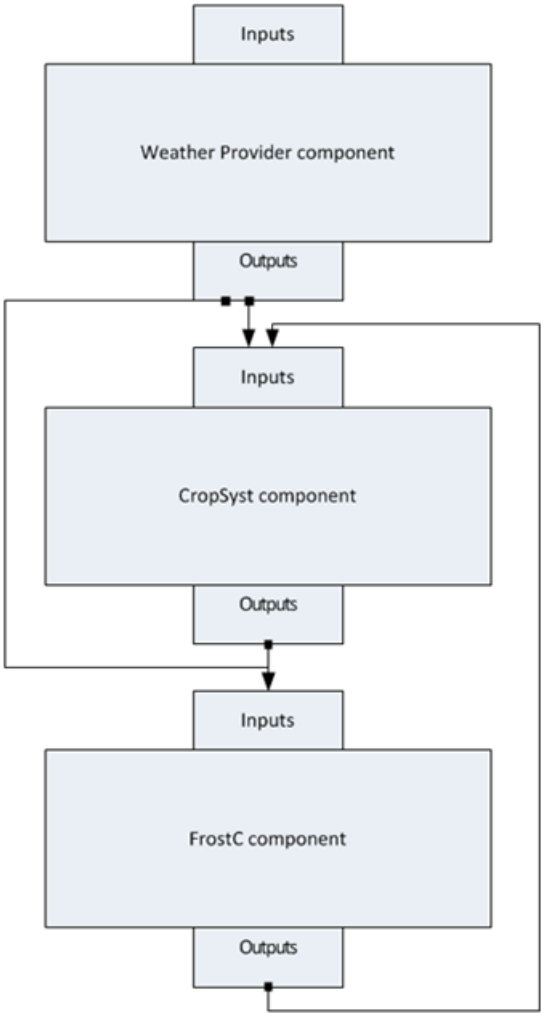
To compose two components means to execute them in the right order, as well as to set the first component output as the second component input.

This connection between the components is defined in a so-called **link**.

Each component has its inputs, outputs and parameters. Each component has a **`run time data`** property which contains all the domain classes that the component must access. The `run time data` values of a component are read/written by other components through the link classes.

The following shows a composition of three components (`ModelRunner`):

Figure 1 Model Runner composed by three components



Schema description:

The arrows represent the **links** between the components' inputs and outputs:

- The **Weather Provider component** is connected to the **CropSyst component** and to the **FrostC component**.
- The **CropSyst component** is connected to **FrostC component**.

- The **FrostC component** is connected to the **CropSyst component**.

Purposes of the simulation components' composition

The composition of simulation components manages:

- The order of execution of the components.
- The links between the components.
- The inputs of the overall composition, as the union of the properties of each component. The inputs of a component are not considered as the inputs of the overall model if they are outputs of another component.
- The outputs and the parameters of the overall composition, as the union of the properties of each component.
- The entry point method to call the execution of the overall composition.

Related topics:

- "Language reference" on page 9

Language reference

3

This chapter is aimed to describe how the Composition layer is coded.

The topics are organized as follows:

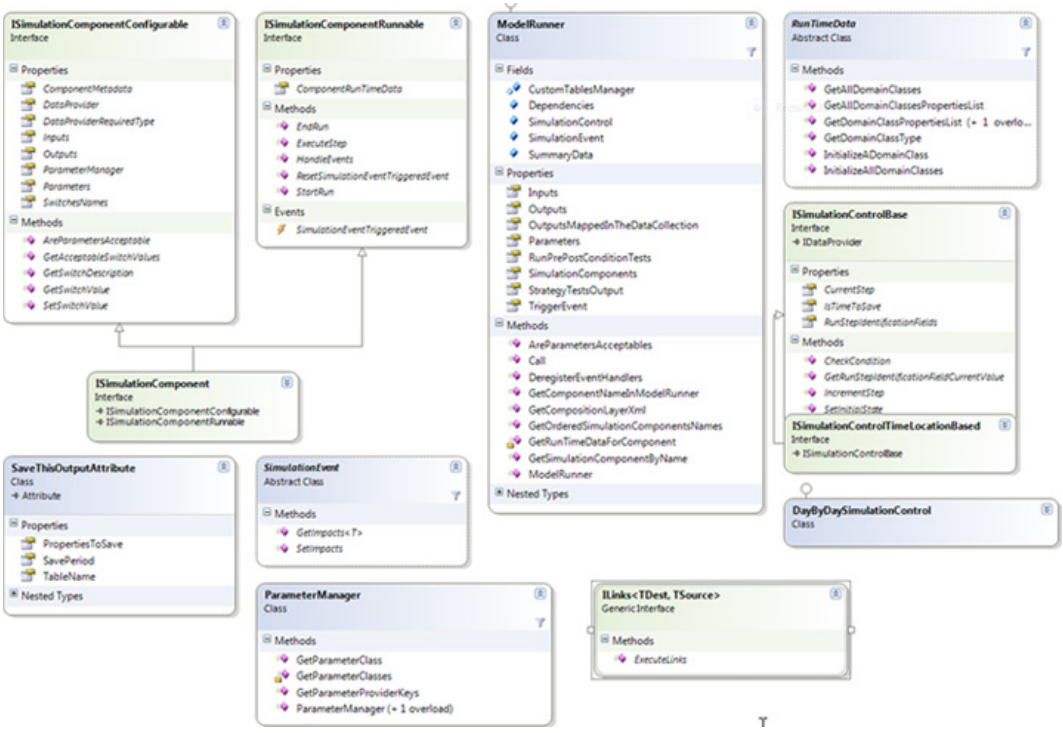
Classes description

The composition layer is coded in two C# code libraries:

- **EC.JRC.MARS.CompositionLayer.Core** library: it contains all the core classes and the interfaces of the Composition Layer.
- **EC.JRC.MARS.CompositionLayer.PreBuiltComponents** library: it contains a set of components that are useful to create a weather/soil/agro management based model.

The figure below shows the main classes of the composition layer library that are described in this topic:

Figure 2 Diagram class of the Composition layer library



For further information, see:

- “iModelRunner” on page 11
- “ISimulationControlBase interface” on page 12

iModelRunner

The **`ModelRunner`** is the composition of several components. By calling its Call method the consumer calls the components in the right order and obtains a `ModelOutput` object. The `ModelRunner` executes a cycle starting from a start step to a final step (usually, these steps are time steps). For each step the `ModelRunner` calls the `ExecuteTimeStep` method of each component.

The `ModelRunner` is the main class of the composition layer.

It represents the modelling solution, composed by a certain number of simulation components, and manages the link/dependencies between the components.

The `ModelRunner` owns an ordered set of `ISimulationComponents` and calls them in the proper order.

To execute the modelling solution, the `Call` method must be called. The `ModelRunner` puts together the inputs/outputs/parameters of each simulation component to expose the inputs/outputs/parameters of the whole modelling solution.

Before calling a simulation component, the `ModelRunner` executes the links towards the component, to fill the inputs of the component by using the outputs of the other components, previously executed.

A link is defined by an instance of the `ILinks<RunTimeData, RunTimeData>` class. Moreover, the `ModelRunner` contains the info about the mutual dependencies of the components and the logics to save the output of the model (a `DataCollection` instance).

To insert a component in the `ModelRunner` (as well as in the modelling solution) the method `InsertSimulationComponent` must be called.

Each component is uniquely identified, within the `ModelRunner`, by a component name, which is defined by the `ModelRunner` when the component is inserted.

ISimulationControlBase interface

The **`ISimulationControlBase interface`** defines the initial state of the cycle, the termination condition, the increment between a step of the cycle, and the next. It also defines the fields to identify the step of the cycle.

The most common `ISimulationControlBase` realization is the `DayByDaySimulationControl` which defines a cycle from a date to another, being the incremental step one day. The `ModelRunner` owns one instance of the `ISimulationControlBase` interface.

Its inputs, outputs and parameters are the collections of the inputs, outputs and parameters of the different components. It contains the collection of the `ISimulationComponents` which are registered in the correct order of execution. It also manages the events thrown by the simulation components, triggering events that the other components can listen.

A **simulation component** is the elementary unit of code in the composition layer. A simulation component represents a piece of software that can be composed with others to create a modelling solution. The composition is defined in a concrete instance of the model runner class.

This interface is the union of two interfaces, representing the two ways to handle a simulation component.

ISimulationComponentConfigurable interface

This interface should be used by an external application to configure, edit the behavior and analyze a simulation component.

- It defines the methods and properties to configure a simulation component and to analyze its inputs/outputs/parameters. The user can change the values of these variables but not the collection of variables itself: the list of inputs, outputs and parameters are defined during the component's creation.
- It has a **`ParameterManager`** property, which represents the logics to provide the values of the parameters. The `ParameterManager` class contains the logics to return the parameter classes of the component.
- It has a **`DataProvider`** property, which represents the logics to provide the external data needed by the component from an external source (e.g., a form of persistence, like a DB). The data provider must implement the `IDataProvider` interface. This interface is defined in the model layer. The type of the data provider instance must be the

one specified by the `DataProviderRequiredType` property of the simulation component.

- It contains the methods to manage the switches of the component: methods to get the list of the switches, the list of the acceptable switch values, in order to set/get the value of the switches. A switch is a handle for the user to change the internal behavior of the component. For example, the user can change between different approaches that the component makes available to calculate the same variable. The switches are defined by the component's creator. He also defines the acceptable values for each switch.

ISmulationComponentRunnable

This interface defines the set of methods that allow the execution of a simulation component. This interface is to be used only by the `ModelRunner`.

It contains:

- The methods for the execution cycle: an initialization method (**`StartRun`**), a step-by-step execution method (**`ExecuteTimeStep`**), and a finalization method (**`EndRun`**).
- A **`HandleEvents`** method to handle the events triggered by the other components.

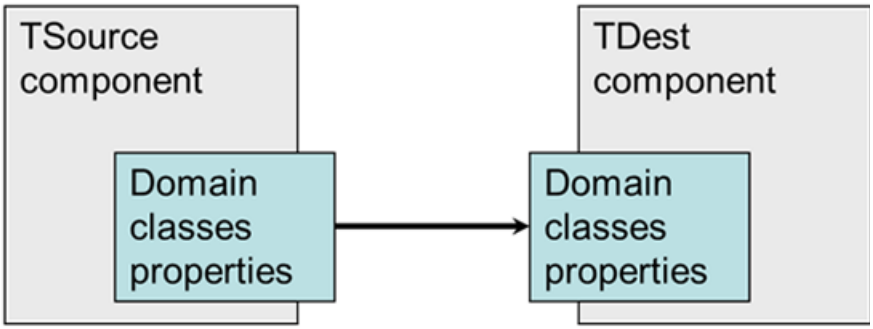
ILink<TDest,TSource> interface

This interface defines the link between two domain classes.

- **`TDest`** is the type of the destination component's run time data (containing the domain classes of the destination component).
- **`TSource`** is the type of the source component's run time data (containing the domain classes of the source component).

To execute the link means to set some properties values (at least one) from one of the domain classes of the `TSource` runtime data to the properties of one domain class of the `TDest` runtime data.

Figure 3 Connection between two domain classes of two linked components



SaveThisOutput attribute

This attribute defines the domain classes (or the domain class properties) to be included in the model output structure: the **`DataCollection`** object (which is a lightweight version of the standard .NET `DataSet` object. Basically, it is a collection of table objects).

The modelling solution developer must tag with this attribute all the domain classes he wants to include in the output of the model.

Example

Save three properties:

- `DevelopmentStageCode`
- `AbovegroundBiomass`
- `Yield`

of the domain class **`States`** in the output structure, in a table called `CropSystPot1_States`:

```
[SaveThisOutput(SavePeriod =
   SaveThisOutputAttribute.SavePeriodEnum.EveryS
   tep, TableName = "CropSystPot1_States",
   PropertiesToSave = new[]
   {"DevelopmentStageCode",
   "AbovegroundBiomass", "Yield"})]
public States States;
```

SimulationEvent

The `SimulationEvent` is an auxiliary class to store data related to the events triggered by the components.

Related topics:

- “ModelRunner execution cycle” on page 16
- “Pre-built components” on page 18
- “CLIC application” on page 19

ModelRunner execution cycle

This topic provides a description of the steps you go through for the execution of a Model Runner.

The components' methods are executed in the order defined by the `SimulationComponents` list, which is the order in which the components were inserted in the model (by the `InsertSimulationComponent` method).

Step 1 - Initialization of the model

1. Each component is registered as event handler.
2. Each component is registered as event source.
3. Each component's domain classes are initialized.
4. Each component's parameter provider is executed to retrieve the component's parameters list. The actual values of the parameters (`VarInfo.CurrentValue`) are written in the corresponding `VarInfo` of the `ModelRunner.Parameters` property.
5. Each component's `StartRun` method is executed

Step 2 - Run of the model

1. The simulation control is set in its initial state by calling the `ISimulationControlBase.SetInitialState` method
2. While the simulation control condition is checked (by calling the `ISimulationControlBase.CheckCondition` method) - e.g., for each time step of the simulation, from start time to end time - execute the step of the simulation. For each component:
   a. Link the outputs of the dependency components (if they exist) to the input of the current component.
   b. Execute the `ExecuteStep` method of the component.
   c. Collect the outputs of the component at the current step.
   d. Pass to the next component.
   e. When all components are executed, increment the simulation control step, by using the `ISimulationControlBase.IncrementStep` method.
3. When the cycle finishes (`ISimulationControlBase.CheckCondition() ==false`), collect the final outputs of the simulation

4 Returns the `DataCollection` containing partial outputs at each step and the final outputs of the simulation.

Step 3 - End of the model

1 The `EndRun` method of each component is executed.

Related topics:

- “Classes description” on page 10
- “Pre-built components” on page 18
- “CLIC application” on page 19

Pre-built components

A set of components is coded in the library **`EC.JRC.MARS.CompositionLayer.PreBuiltComponents`** to offer to the modelling solutions developer a set of functionalities of common use.

These pre-built components can be useful for creating many agronomic modelling solutions.

Weather provider component

(class `WeatherProviderSimulationComponent`)

This component retrieves weather data from a data provider of type `IWeatherProviderBase`. This component can be included in modeling solutions that require weather data.

Soil data provider component

(class `SoilDataProviderSimulationComponent`)

This component retrieves soil data from a data provider of type `ISoilDataProviderBase`. This component can be included in modeling solutions that require soil data.

Agro management data provider component

(class `AgroManagementSimulationComponent`)

This component retrieves agro management data from a data provider of type `IAgromanagementProviderBase`. This component can be included in modeling solutions that require agro management data.

For further details about the agro management and its data formats please refer to the Agro management documentation.

Related topics:

- “Classes description” on page 10
- “ModelRunner execution cycle” on page 16
- “CLIC application” on page 19

CLIC application

CLIC (Composition Layer Interactive Coder) is a desktop application aimed to assist modelers in creating a composition layer C# project.

CLIC allows to easily create a composition of existing components and to create new components starting from the model layer's strategies.

Moreover, by using CLIC, single components can easily be extracted from an existing modeling solution to be used in another modeling solution.

It also allows exploring an already compiled project containing a model runner and drawing a graph of the structure: a list of components that shows how components are linked one another (the graphic metaphor is at early stage of development).

Tip:

For further information on the CLIC application, please refer to the CLIC User Guide.

Related topics:

- “Classes description” on page 10
- “ModelRunner execution cycle” on page 16
- “Pre-built components” on page 18