VisualAivika Manual

Creating External Modules

David E. Sorokin <davsor@mail.ru>
(Yoshkar-Ola, Mari El, Russia)

July 11, 2026

Contents

Introduction

VisualAivika is a visual simulation tool destined for System Dynamics and Discrete Event Simulation. It allows writing external modules in .NET

so that these modules could be used within simulation. This document *VisualAivika Manual: Creating External Modules* is devoted to that how you can create your own external modules.

It is worth noting that VisualAivika (namely, its Equation Compiler version) allows you to export your simulation models as .NET applications. These modules are based on IronAivika and a small additional run-time library, which sources are available in the source form (under the GPL3 license). Even if you will add your own external modules then all the code will be available. It is important that such models exported as the applications can be built and then launched without VisualAivika. VisualAivika can help you to create and test such models, but then the applications can be independent from VisualAivika itself.

It is supposed that the reader is familiar with the VisualAivika modeling language and with the IronAivika documentation.

1 External Modules

In this chapter we will write a small external module that will delay the input `Block` computation for one unit of modeling time. Also it will write in the console a log message before the delay and after it.

To see the log messages on Windows, you will need to export the corresponding simulation model as a .NET application. On Linux, you can just launch VisualAivika from the console.

We have to write the definition of our new external module. VisualAivika recognizes such definitions, which are required for integrating the external modules in the simulation.

Let our external module definition will have name `M`. It must be defined as the *Auxiliary* item on the diagram of VisualAivika.

```
M =
  [<assembly("lib/DelayBlockModule.dll")>]
  [<class(VisualAivika.Demo.DelayBlockModule)>]
  extern module (A: Block) {
      Y: Block;
  };
```

In the Equation Editor you should define that part of the equation that follows the assignment sign.

Here we say that the external module will be compiled in the *DelayBlockModule.dll* assembly, which will be located in the *lib* directory relative to that directory, where our VisualAivika simulation model will be saved in. One external module definition may have many `assembly` attributes. All such assemblies will be linked to the simulation model. It is possible to specify absolute file names too.

The `class` attribute specifies the template for the .NET classes that we will define soon. This is not a single .NET class type. This is a template based on which VisualAivika will find the names of the corresponding .NET classes. There will be a few such classes.

Also we say that our module takes the input block and returns a new block as its `Y` property. The input parameter name `A` will be used in our F# code below. The same is true for the output property `Y`.

It makes sense to make three notes.

At first, the start time of simulation is a separate stage in VisualAivika. Every simulation entity must have its initial form, whether this is an integral, or block, or anything else. For example, the integral has the initial value by definition.

Regaring any `Block` computation, its initial form must be always defined as the `Block.terminate` computation. Regarding any `Stream` computation, its initial form is always the `Stream.empty` computation.

When using blocks and streams as parameters in your external modules, you will need to return namely these computations as initial forms, although VisualAivika can even automatically substitute them in some important cases. Therefore, it will be useful to follow always these rules.

At second, VisualAivika supports three integration methods: Euler's method, the Runge-Kutta method of the 2nd and 4th orders. We must support all them.

At third, VisualAivika supports an ability to redefine some constant parameters within simulation. VisualAivika 9 does not expose it in GUI. It literally means for us that we have still to support the `Correct` method as you will see soon, even if we do not see it in GUI. The purpose of this method is to check the array dimensions after the parameter update, for example.

Now it is time to show the corresponding F# code.

```
#nowarn "40"
```

```
namespace VisualAivika.Demo

open System

open Simulation.Aivika
open Simulation.Aivika.Blocks
open Simulation.Aivika.Results

open VisualAivika.Lang.Simulations

type DelayBlockModuleInputInitial () =

    let mutable a:
        Block<Transact<Map<string, obj>>, unit> =
            Block.terminate

    member x.AInitial
        with get (): Block<Transact<Map<string, obj>>, unit> = a
        and set (v: Block<Transact<Map<string, obj>>, unit>) = a <- v

type DelayBlockModuleInput () =

    let mutable a:
        Block<Transact<Map<string, obj>>, unit> =
            Block.terminate

    member x.A
        with get (): Block<Transact<Map<string, obj>>, unit> = a
        and set (v: Block<Transact<Map<string, obj>>, unit>) = a <- v

type DelayBlockModule () =

    let mutable y = Block.terminate

    member x.YInitial:
        Block<Transact<Map<string, obj>>, unit> =
            Block.terminate

    member x.Y:
        Block<Transact<Map<string, obj>>, unit> =
            y

    member x.PreInitialize (run: SimulationRun,
                            ps: SimulationParameterCollection,
                            input: DelayBlockModuleInputInitial) = ()
```

```
    member x.Initialize (run: SimulationRun,
                         ps: SimulationParameterCollection,
                         input: DelayBlockModuleInput) =

        y <- Block.arrc (fun (transact: Transact<Map<string, obj>>) ->
            proc {

                let! t1 = Dynamics.time |> Dynamics.lift
                printfn "t = %f: before 1-unit delay" t1

                do! Proc.hold 1.0

                let! t2 = Dynamics.time |> Dynamics.lift
                printfn "t = %f: after 1-unit delay" t2

                return! Block.run transact input.A
            })

    member x.Correct (ps: SimulationParameterCollection,
                      p: SimulationPoint) = ()

    member x.Iterate (p: SimulationPoint) = ()

    interface IDisposable with
        member x.Dispose () = ()

type DelayBlockModuleEulerProcess () =
    inherit DelayBlockModule ()

type DelayBlockModuleRK2Process () =
    inherit DelayBlockModule ()

type DelayBlockModuleRK4Process () =
    inherit DelayBlockModule ()
```

At the start time of simulation we prepare the input parameters with help of the `DelayBlockModuleInputInitial` class type. Then VisualAivika creates an instance of the `DelayBlockModule` class type and calls the `PreInitialize` method. This is the state machine for our module. As we will see later, some properties are treated differently for initial and ordinary forms of the external module parameters.

Also all initial properties must have word *Initial* as a suffix in their names. This is the naming convention that allows VisualAivika to distin-

guish the initial values from the ordinary iterative ones.

After the pre-initialization is complete, the input parameters are prepared with help of the `DelayBlockModuleInput` class type. As we just mentioned, some properties can be treated differently in comparison to the initial form. Then VisualAivika uses the same instance of the external module and calls already the `Initialize` method.

Also we have to provide three different implementations of the external modules, since VisualAivika supports three different integration methods for the system of differential equations. If you do not use the differential equations then all three implementations can have the same code base.

It is worth special noting that you should provide all the mentioned classes with the same naming convention as it was provided in the example. Here all classes have the same prefix in their names, which was exactly defined in the `class` attribute. This is the naming convention expected by VisualAivika.

Then we can use the defined external module in VisualAivika like this

```
M1 = M(Block.advance(10) >>> Block.terminate);
Y1 = M1->Y;
```

Here `Y1` is a `Block` computation which is returned from the external module. One external module may have multiple output parameters.

2 External Functions

The *External Function* is a particular case of the external module, where there is only one output propery and its name is exactly `Result` on level of the F# code. There are no other differences on level of the F# code. But the VisualAivika definition is different.

For example, the same module from the previous section could be redefined as a function, for the module has only one output property. Then we could write the following definition in VisualAivika.

```
F =
  [<assembly("lib/DelayBlockFunction.dll")>]
  [<class(VisualAivika.Demo.DelayBlockFunction)>]
  extern fun (A: Block): Block;
```

Regarding the implementation of the `DelayBlockFunction` type class, we must rename the `Y` property as `Result` along with the corresponding class types, and this is all we have to do.

But we receive directly the function result on level of VisualAivika. There is no need to access the property anymore.

```
Y1 = F(Block.advance(10) >>> Block.terminate);
```

It looks like that the `F` external function would be built-in in VisualAivika.

3 Data Types

Below is described the correspondence between the VisualAivika and F# data types. The focus is on the discrete event simulation entities such as blocks and storages. The support of differential equations may look strange and hard to use, but it is designed in such a way so that VisualAivika models could be used as external modules in the future. This is actually related to that how the Equation Compiler generates the code for integrating differential equations.

Here we suggest that the following namespaces are open:

```
open Simulation.Aivika
open Simulation.Aivika.Blocks
```

The following Table 1 shows the correspondence for input parameters which the external module may have. There is a similar correspondence for output parameters too.

Table 1: The correspondence between the VisualAivika and F# data types for input parameters.

VisualAivika	The Input F# Property Type
unit	obj *(but the value is `null`)*
int	int
float	float
string	string

SamplingStats<int>	SamplingStats<int>
SamplingStats<float>	SamplingStats<float>
TimingStats<int>	TimingStats<int>
TimingStats<float>	TimingStats<float>
Dynamics<int>	int *(for the initial form only)*
Dynamics<float>	float *(for the initial form only)*
Dynamics<int>	delegate of unit -> int
Dynamics<float>	delegate of unit -> float
Dynamics<SamplingStats<int>>	SamplingStats<int> *(for the initial form only)*
Dynamics<SamplingStats<float>>	SamplingStats<float> *(for the initial form only)*
Dynamics<SamplingStats<int>>	delegate of SamplingStats<int>
Dynamics<SamplingStats<float>>	delegate of SamplingStats<float>
Dynamics<TimingStats<int>>	TimingStats<int> *(for the initial form only)*
Dynamics<TimingStats<float>>	TimingStats<float> *(for the initial form only)*
Dynamics<TimingStats<int>>	delegate of TimingStats<int>
Dynamics<TimingStats<float»	delegate of TimingStats<float>
Block	Block<Transact<Map<string, obj>>, unit>
Stream	Stream<Arrival<Map<string, obj>>>
Signal	Signal<Arrival<Map<string, obj>>>
Queue	Lazy<Queue>
Facility	Lazy<Facility<Map<string, obj>>>
Storage	Lazy<Storage>
MatchChain	Lazy<MatchChain>
Ref<int>	Lazy<Ref<int>>
Ref<float>	Lazy<Ref<float>>
Ref<SamplingStats<int>>	Lazy<Ref<SamplingStats<int>>>
Ref<SamplingStats<float>>	Lazy<Ref<SamplingStats<float>>>
Ref<TimingStats<int>>	Lazy<Ref<TimingStats<int>>>
Ref<TimingStats<float>>	Lazy<Ref<TimingStats<float>>>

If you have the integral, or random function, or any other numerical expression `e` then you can receive the corresponding initial value in VisualAivika by calling the `init(e)` operator. Such an initial value will be calculated at the start time and considered to be constant within the current simulation run, and it can be passed in to the external module as the value of `int` or `float` data types.

Regarding the integral itself, or random function, or any numerical expression dependent on the modeling time, they should be passed in to the external module as the value of the `Dynamics<float>` data type. This name is related to that how they are treated in the Interpreter version of VisualAivika. But the Equation Compiler version of VisualAivika treats such values as .NET delegates, which can be called in every integration time point.

Thus, the integral is `Dynamics<float>`. So are many random functions. Any other numerical time-dependent expression is `Dynamics<float>` too, and so on.

The list of data types which can be used for output parameters is shorter. The list is shown in Table 2.

Table 2: The correspondence between the VisualAivika and F# data types for output parameters, which are returned as .NET properties

VisualAivika	The Output F# Property Type
unit	obj *(but the value is `null`)*
Dynamics<int>	int
Dynamics<float>	float
Dynamics<SamplingStats<int>>	SamplingStats<int>
Dynamics<SamplingStats<float>>	SamplingStats<float>
Dynamics<TimingStats<int>>	TimingStats<int>
Dynamics<TimingStats<float>>	TimingStats<float>
Block	Block<Transact<Map<string, obj>>, unit>
Stream	Stream<Arrival<Map<string, obj>>>
Signal	Signal<Arrival<Map<string, obj>>>

They correspond to the .NET properties too. For example, the `int` and `float` properties related to the `Dynamics` computation are called in every integration time point.

If you want to return a constant value from the external module then you actually have to return the `Dynamics` computation, which you can apply the `init` operator to.

4 Unit Data Type

The treating of the `unit` data type is special, for VisualAivika generates the C# code, where there are some restrictions related to using this data type. But we do need this type to perform some side effect like resetting the statistics at the specified modeling time.

While the ordinary output property is defined as the C# property of some similar type, the unit property must have the resulting type `obj` like this:

```
type UnitFunctionInputInitial () = class end

type UnitFunctionInput () = class end

type UnitFunction () =

    member x.ResultInitial: obj = null

    member x.Result: obj = null

    member x.PreInitialize (run: SimulationRun,
                            ps: SimulationParameterCollection,
                            input: UnitFunctionInputInitial) = ()

    member x.Initialize (run: SimulationRun,
                         ps: SimulationParameterCollection,
                         input: UnitFunctionInput) = ()

    member x.Correct (ps: SimulationParameterCollection,
                      p: SimulationPoint) = ()

    member x.Iterate (p: SimulationPoint) = ()

    interface IDisposable with
        member x.Dispose () = ()

type UnitFunctionEulerProcess () =
    inherit UnitFunction ()

type UnitFunctionRK2Process () =
    inherit UnitFunction ()

type UnitFunctionRK4Process () =
    inherit UnitFunction ()
```

Here we define the VisualAivika external function with the `Result` property of the resulting type `obj` that has a single possible value `null`.

The same approach works if we define output properties of the `unit` type for external modules, not only functions.

Regaring the input parameter of type `unit`, we should define the corresponding setter with the parameter of the `obj` type.

```
type TestModuleInputInitial () =

    let mutable a: obj = null

    member x.AInitial
        with set (v: obj) = a <- v
```

Such constraints exist due to limitations of that how the C# language treats the `void` type and how F# integrates the `unit` type in C#.

5 Arrays

The arbitrary external module instances can be used within arrays in a natural way. We can create an array of the external module instances.

For example, if we take the external module definition from section 1 then we could define the following arrays.

```
M1 = [ M(Block.advance(10) >>> Block.terminate) | i <- 1..3 ];
Y1 = [ M1[i]->Y | i <- 1..3 ];
```

But there is a more difficult case when we pass the arrays in input parameters, or when we return the arrays as output parameters. In such a case, the F# code treats such parameters as arrays of the corresponding data type, where the indices are started from zero on level of the F# code. But the start indices can be arbitrary on level of VisualAivika.

5.1 Input Arrays

Let us write a simple external module that takes the array of `Block` computations and launch them in parallel for every input transcat. There is no such corresponding built-in function in VisualAivika yet.

The VisualAivika interface is as follows.

```
P =
  [<assembly("lib/ParBlockFunction.dll")>]
  [<class(VisualAivika.Demo.ParBlockFunction)>]
  extern fun (Xs: Block [..]): Block;
```

If the array parameter was two-dimensional then its type would be `Block [.., ..]`, and so on. The F# type would be an array of arrays of `Block` computations. The start index of the array is erased. On level of the F# code, all array dimensions are started with zero.

Below is the F# code that corresponds to the case of one-dimensional arrays.

```
#nowarn "40"

namespace VisualAivika.Demo

open System

open Simulation.Aivika
open Simulation.Aivika.Blocks
open Simulation.Aivika.Results

open VisualAivika.Lang.Simulations

type ParBlockFunctionInputInitial () =

    let mutable xs:
        Block<Transact<Map<string, obj>>, unit> array = [| |]

    member x.XsInitial
        with set (v: Block<Transact<Map<string, obj>>, unit> array) =
            xs <- v

type ParBlockFunctionInput () =

    let mutable xs:
        Block<Transact<Map<string, obj>>, unit> array = [| |]

    member x.Xs
        with get(): Block<Transact<Map<string, obj>>, unit> array =
            xs
        and set (v: Block<Transact<Map<string, obj>>, unit> array) =
            xs <- v
```

```
type ParBlockFunction () =

    let yInitial = Block.terminate

    let mutable y = Block.terminate

    member x.ResultInitial:
        Block<Transact<Map<string, obj>>, unit> =
            yInitial

    member x.Result:
        Block<Transact<Map<string, obj>>, unit> =
            y

    member x.PreInitialize (run: SimulationRun,
                            ps: SimulationParameterCollection,
                            input: ParBlockFunctionInputInitial) = ()

    member x.Initialize (run: SimulationRun,
                         ps: SimulationParameterCollection,
                         input: ParBlockFunctionInput) =

        y <- Block.arrc (fun transact ->
            proc {

                let comps = [
                    for x in input.Xs do
                        yield Block.run transact x
                ]

                return! Proc.par_ comps
            })

    member x.Correct (ps: SimulationParameterCollection,
                      p: SimulationPoint) = ()

    member x.Iterate (p: SimulationPoint) = ()

    interface IDisposable with
        member x.Dispose () = ()

type ParBlockFunctionEulerProcess () =
    inherit ParBlockFunction ()

type ParBlockFunctionRK2Process () =
```

```
    inherit ParBlockFunction ()

type ParBlockFunctionRK4Process () =
    inherit ParBlockFunction ()
```

Here we run the corresponding `Block` computations in parallel and then wait for their completion.

Given the `M` external module is defined as it was in section 1, we can write in VisualAivika

```
S1 = P([ M1[i]->Y | i <- 1..3 ])
```

Here `S1` becomes a `Block` computation. When being applied, the `S1` block launches all `Y` blocks from the array in parallel.

5.2 Output Arrays

Output array parameters are similar to input ones. Only now we have to define the start indices for every dimension in a syntax similar to

```
C =
  [<assembly("lib/BlockCloneFunction.dll")>]
  [<class(VisualAivika.Demo.BlockCloneFunction)>]
  extern fun (A: Block, N: int): Block [1..];
```

If we returned a three-dimensional array as output then we could write something similar to `Block [1.., 10.., 0...]` with the start indices 1, 10 and 0, respectively.

Let the `C` external function clones the input block as many copies as the `N` parameter specifies.

We can write this external function in the following way.

```
#nowarn "40"

namespace VisualAivika.Demo

open System

open Simulation.Aivika
open Simulation.Aivika.Blocks
open Simulation.Aivika.Results
```

```
open VisualAivika.Lang.Simulations

type BlockCloneFunctionInputInitial () =

    let mutable a:
        Block<Transact<Map<string, obj>>, unit> =
            Block.terminate

    let mutable n: int = 0

    member x.AInitial
        with get (): Block<Transact<Map<string, obj>>, unit> = a
        and set (v: Block<Transact<Map<string, obj>>, unit>) = a <- v

    member x.NInitial
        with get (): int = n
        and set (v: int) = n <- v

type BlockCloneFunctionInput () =

    let mutable a:
        Block<Transact<Map<string, obj>>, unit> =
            Block.terminate

    let mutable n: int = 0

    member x.A
        with get (): Block<Transact<Map<string, obj>>, unit> = a
        and set (v: Block<Transact<Map<string, obj>>, unit>) = a <- v

    member x.N
        with get (): int = n
        and set (v: int) = n <- v

type BlockCloneFunction () =

    let mutable ysInitial = [| |]

    let mutable ys = [| |]

    member x.ResultInitial:
        Block<Transact<Map<string, obj>>, unit> array =
            ysInitial
```

```
    member x.Result:
        Block<Transact<Map<string, obj>>, unit> array =
            ys

    member x.PreInitialize (run: SimulationRun,
                            ps: SimulationParameterCollection,
                            input: BlockCloneFunctionInputInitial) =

        ysInitial <- [|
            for i in 1 .. input.NInitial do
                yield Block.terminate
        |]

    member x.Initialize (run: SimulationRun,
                         ps: SimulationParameterCollection,
                         input: BlockCloneFunctionInput) =

        ys <- [|
            for i in 1 .. input.N do
                yield Block.arrc (fun transact -> proc {
                    return! Block.run transact input.A
                })
        |]

    member x.Correct (ps: SimulationParameterCollection,
                      p: SimulationPoint) = ()

    member x.Iterate (p: SimulationPoint) = ()

    interface IDisposable with
        member x.Dispose () = ()

type BlockCloneFunctionEulerProcess () =
    inherit BlockCloneFunction ()

type BlockCloneFunctionRK2Process () =
    inherit BlockCloneFunction ()

type BlockCloneFunctionRK4Process () =
    inherit BlockCloneFunction ()
```

The stated above `C` and mentioned earlier `P` external functions can work in tandem like this

```
S1 = P(C(M1->Y, 3));
```

Here we create 3 branches of the input block computation `M1->Y` and then treat all them as the single `Block` computation.

It is worth noting that the `C` external function returns an array, for which we specified the start index. Also we can always receive the array ranges by applying the `low` and `high` functions.

```
Cs1 = C(M1->Y, 3);
Bs1 = [ Cs1[i] | i <- low(Cs1) .. high(Cs1) ];
```

5.3 Nested Arrays

Also VisualAivika allows us to create an array of external module instances, where such instances could have array parameters in its turn.

We could define something like that, whatever it means.

```
K = 2;
M1 = [ M(Block.terminate) | k <- 1..K ];
S1 = [ P(C(M1[k]->Y, 3)) | k <- 1..K ];
```

6 Resume

Thus, VisualAivika allows us to build scalable simulation models with help of the approach of so called external modules. We can embed the arbitrary .NET code in our simulation models. Moreover, we can use arrays to replicate the data structures.