Aivika Overview: One Functional Approach to Simulate Hybrid Models

David Sorokin david.sorokin@gmail.com, Yoshkar-Ola, Russia, 2009 Nov - 2010 Sep

Below is represented one approach to simulate hybrid discrete-continuous models. I invented it in the course of my studying the functional programming. It is a very interesting fusion of such different fields, the functional programming and simulation. The result of this my activity is an extensible modeling library Aivika accessible by the following address: http://sourceforge.net/projects/aivika

Dynamic Process as Computation

In the heart of the new approach I offer is a computation that represents a dynamic process varying in time. The dynamic process is just a function of time that returns values of arbitrary type. In the F# programming language such a process can be expressed as polymorphic type `Dynamics`:

```
type Dynamics<'a> = Time -> 'a
```

In other words any computation of the `Dynamics` type is a function of time. Actually, in the Aivika modeling library the `Time` type is more detailed and it also contains some auxiliary information such as the current integration node for the Runge-Kutta method, if any.

Type `'a` in the definition means any arbitrary type. For example, `Dynamics<float>` is a function that computes floating point numbers. `Dynamics<int>` is a computation that returns integers in the time points. Finally, `Dynamics<Dynamics<'b>>` is a function that returns in time points other functions each of them returns already in its time points some values of the specified type `'b`, i.e. a dynamic process of nested dynamic processes. This is indeed a very flexible abstraction.

The key idea is that the System Dynamics model, Discrete Event Simulation (DES) and the Agent-based model can all be reduced ultimately to a computation of type `Dynamics<'a>`, i.e. a dynamic process, whatever the source model would be. On intuitive level it sounds reasonable. The question is *how* to write it in code. If we can do it, and we can, in a unified way then we can build and run hybrid discrete-continuous simulations. Let's consider how such a conversion can be performed.

System Dynamics

A conversion of the System Dynamics model to the `Dynamics` computation is rather straightforward. Most variables can be represented as values of type `Dynamics<float>`, i.e. some functions of time that return floating point numbers in time points. Moreover, the standard functions provided by such tools as Simtegra MapSys, Vensim and ithink can be replaced with their counterparts that accept the computations as arguments and return new computations as the result.

Perhaps the most difficult case is the integral function. Here is its definition in Aivika:

```
val integ: Lazy<Dynamics<float>> -> Dynamics<float> -> Dynamics<float>
```

This function gets a derivative as a delayed computation (wrapped in the `Lazy` type) and the initial value of the integral at the start time. It returns a new computation that represents the integral value. This new computation has type `Dynamics<float>`. In the functional programming such a function is called a *combinator*.

Similarly, we can overload the standard arithmetic operators and functions such as `sin` and `cos`. All this allows us to define the System Dynamics model very intuitively and declaratively in F#:

```
let rec a = integ (lazy (-ka * a)) (eta 100.0)
and b = integ (lazy (ka*a - kb*b)) (eta 0.0)
and c = integ (lazy (kb*b)) (eta 0.0)
and ka = 1.0
and kb = 1.0
```

Here values `a`, `b` and `c` have type `Dynamics<float>`, i.e. they are computations that return floating-point numbers. Their equations exactly correspond to the following system written in the Berkelley-Madonna language:

```
d/dt (A) = - ka*A
d/dt (B) = ka*A - kb*B
d/dt (C) = kb*B
init A = 100
init B = 0
init C = 0
ka = 1
kb = 1
```

The difference is the former model defines values in the general-purpose high-level programming language F#. As it will be shown, it gives us fantastic opportunities to use external functions within the simulation. There are no actually bounds between the simulation and the external code. They all exist in the same space.

The approach of using computations is so productive that we can easily express new functions using already defined. Here is a definition of the `smooth3` function in F#:

```
let smooth3 (x: Dynamics<float>) (t: Lazy<Dynamics<float>>) =

    let rec y = integ (lazy ((s1 - y) / (t.Value / 3.0))) x
    and s1 = integ (lazy ((s0 - s1) / (t.Value / 3.0))) x
    and s0 = integ (lazy ((x - s0) / (t.Value / 3.0))) x
    in y
```

We define three stocks and return the first one as the function result. The `t` parameter is defined lazy. Therefore we use its property `Value` to get the corresponded computation. In all three cases of applying the `integ` function we create totally new computations. Also we can say that the `smooth3` function has no explicit side-effect. Its definition is very declarative and high-level as it would be Vensim's macro, for example.

To complete the picture, Aivika provides built-in computations for the main simulation parameters:

```
val starttime: Dynamics<float>
val stoptime: Dynamics<float>
val dt: Dynamics<float>
val time: Dynamics<float>
```

This record means that all four bult-ins are computations, i.e. functions of time that return floating-point numbers. The `starttime`, `stoptime` and `dt` built-ins return constants in time points. The `time` computation of type `Dynamics<float>` replicates the time point specified as a function argument.

You may ask how these built-in parameters know of their values. It is very simple. These values are passed as a function argument. Earlier I wrote that the `Time` type may contain the additional information. These values are a part of this information that is passed on to every computation of the `Dynamics` type!

It is turned out that we can reduce the System Dynamics model to a set of values each of them represents some computation of the `Dynamics` type. We can do the same for other simulation paradigms as well.

Time-driven DES

The time-driven DES is possibly the simplest case. It is related to that the `Dynamics` computation is actually a *monad*. It means that in F# we can create the corresponded workflow. I called it *the dynamics workflow*. Then using syntax of computation expressions that F# provides and the built-in parameters described above, we can represent a time-driven simulation as some computation of the `Dynamics` type.

The key feature of the computation expressions is an ability to bind arbitrary computations together in a simple declarative form, where we can incorporate almost any F# code between the computations and make them a part of the outer computation. This is one of the main traits inherent in any monad.

```
let machine i =
    dynamics {
        ...
        // the machine is broken
        let! finishUpTime = time
        totalUpTime <- totalUpTime
            + (finishUpTime - !startUpTime)
        ...
    } |> Dynamics.memo0 discrete
```

It is worth noting that the specified time-driven simulation excerpt refers to the `time` built-in parameter that was actually introduced for System Dynamics. Moreover, there is a link in opposite direction. Stated in this example the `discrete` function is related to System Dynamics too. It is turned out that the time-driven simulation and the values defined in the System Dynamics model including built-ins can easily cooperate. Actually, there is no any difference between them from the simulation standpoint. All they are functions of time and have the same generic F# type.

Event-driven DES

The event-driven simulation is no exception. Only now we have to define a special class to represent *the event queue*. In the Aivika modeling library this class is called `Env`. It is implemented on top of the binary heap.

```
type Env =
    new: unit -> Env
    member Run: Dynamics<unit>
    member Dispatch: t:Dynamics<float> * c:Dynamics<unit> -> Dynamics<unit>
    member DispatchD: t:Dynamics<float> * c:(Dynamics<unit -> unit>) -> Dynamics<unit>
```

The `Dispatch` and `DispatchD` methods allow us to register events that will be actuated at the specified time. The first argument of the both methods defines this time. The second argument specifies already an event.

Here the most intriguing thing is that the event is a computation of the `Dynamics` type too! How does it work?

F# is a very high-level language, being general-purpose at the same time. As a result, it supports *closures*. Each time we want to pass some data with the event we actually create a closure.

```
let machine i (e: Env) =
    let rec broken startUpTime =
        dynamics {
            ...
            // the machine is broken
            let! finishUpTime = time
            totalUpTime <- totalUpTime
                + (finishUpTime - startUpTime)
            ...
            // register a new event
            do! e.Dispatch (eta (finishUpTime + repairTime),
                    repaired)
        }
    and repaired =
        dynamics {
            ...
            // register a new event
            do! e.Dispatch (eta (startUpTime + upTime),
                    broken startUpTime)    // closure
        }
    ...
```

In the second case we register a new event which is a result of applying function `broken` to argument `startUpTime`. This result is a computation that remembers the specified argument and then uses it in the body when the total up time is modified.

So, the event is a computation, i.e. a function of time. After the event is raised the event queue calls this event's function passing the current time on. To functionate properly, the event queue must have a driving force that will periodically raise events from the queue. This driving force is a value of the `Run` property of the `Env` class type. Involving this value of the `Dynamics<unit>` type in the simulation, we run the event queue. It is very easy to do this with help of the `do!` keyword somewhere in the calling method.

Here the value of type `Dynamics<unit>` which is returned by the `Run` property of the event queue is worth the reader's attention. The `unit` type has only one possible value denoted as `()` in F#. Therefore `Dynamics<unit>` means a function that performs some side effect and then always returns value `()`. It is amazing but this is also a dynamic process. It means that the driving force of the event queue is just some function that performs some useful work in time points. What is important, this is also the `Dynamics` computation that can be involved in the simulation.

Thus, the events are computations of the `Dynamics` type. Therefore they can use parts of the System Dynamics model and the results of the time-driven simulation. But the event queue must be involved in the main simulation through the `Run` property value.

Process-oriented DES

The process-oriented DES is another kind of beast. Here we deal with control processes that can be suspended at any time and then resumed later. In the functional programming the rule is to apply

continuations to implement such a behavior. In the Aivika modeling library I created a new computation which I called `DynamicsCont`. The corresponded F# workflow is called *the dynamicscont workflow*. `DynamicsCont` is actually a particular case of *the monad transformer* received as a result of mixing the continuations and the `Dynamics` monad.

Here is the simplest form for the new computation:

```
type DynamicsCont<'a> = Dynamics<'a -> unit> -> Dynamics<unit>
```

As before, this type is polymorphic and its parameter `'a` can be an arbitrary type. It is important that any computation in the source `Dynamics` type can be *lifted* to be a new computation of the `DynamicsCont` type.

The new type allows us to suspend a computation at any time. Moreover, we can pass on the rest of the computation, i.e. a continuation, as an event to the `DispatchD` method of the event queue. It means that the computation will be resumed later at the specified time.

Also we can hold any computation suspended, for example, while the resource is not released. It allows us to easily implement the competitive resource acquisition.

Since every computation of the `Dynamics` type can be lifted, we can use it within our new computation of the `DynamicsCont` type. The following example shows how the `time` built-in parameter is lifted and then used.

```
type Machine (e, id) =
    inherit Process (e)
    override x.Activate = dynamicscont {
        while true do
            let! startUpTime = DynamicsCont.lift time
            let upTime = expovariate upRate
            do! x.Hold (upTime)
            ...
    }
```

Here the `Hold` method suspends the current control flow for the specified time interval. Under the hood it takes the continuation of the computation and passes it on as an event to the queue. After the specified up time elapses the event queue raises the event, which ultimately leads to resumption of the `DynamicsCont` computation.

Thus, the process-oriented simulation can be implemented on top of the event queue. Like that how the event queue must be involved in the simulation through the `Run` property value, there are similar methods for the control processes that allow us to incorporate these processes into the main simulation. It is possible due to the fact that any computation of the `DynamicsCont` type can be run as the `Dynamics` computation. Similarly, we can use any computation of the `Dynamics` type such as the values of the System Dynamics model in the process-oriented simulation.

It is worth noting that the `DynamicsCont` workflow is rather slow and it has a sensible overhead. At the same time it makes the process of defining the model very pleasing and attractive as well as less complex. This is a reasonable trade-off.

Agent-based Modeling

The agent-based model uses the state machine. The states can be nested and each agent has its own tree of the states. We can assign to each state a set of handlers that are actuated according to the specified schedule. I think it will not surprise you but these handlers can be represented as computations of the `Dynamics` type. Moreover, the activation and deactivation of the states can be represented as the `Dynamics` computations too.

```
let adopter =
    { new State (agent) with
        member x.Activate = dynamics {
            ...
            // create a timer that will hold while the state is active
            do! x.SetTimerD (dynamics {
                                return expovariate contactRate
                            },
                            dynamics {
                                do! agents.[random 0 (agents.Count - 1)].Buy
                            })
        }
    }
```

Here the adopter value defines a state to which we assign a timer with the specified random time interval and action to perform. We send a `Buy` message to random agents in random intervals that are calculated using the exponential distribution. You may notice that the dynamics workflow is used everywhere.

Since every aspect of agent's behavior is defined in terms of the `Dynamics` computation, we can easily connect the agent with other parts of the main model whether they are defined as the System Dynamics model or discrete event simulation. It is all possible due to the universal nature of the `Dynamics` computation that can represent any dynamic process.

The state handlers need the event queue to functionate. Therefore each agent requires the queue value during its construction. It means that the agent-based modeling is implemented on top of the discrete-event DES in the Aivika modeling library.

Simulation

After we express our input dynamic model as some computation of the `Dynamics` type, we can run the simulation. Such a computation is just a function of time. To receive the results, we can apply this function to an arbitrary set of time values. For example, if we take the Runge-Kutta method then we can create the integration nodes with some small step and then call this function in these nodes. The function will immediately return the simulation results. There is no magic here at all. The magic was in that *how* we constructed that computation.

We usually need to know the simulation results for more than one variable. It is easy to receive using the computation expression syntax or the predefined functions of Aivika. The `Dynamics` type is a monad. Therefore we can combine the results in any way we wish. Then the simulation function will return exactly those results we need. Also we can log the results during the simulation.

Synchronization

Till now I deliberately didn't use word *variable*. Instead I tried to use word *value* to emphasize the fact that the value is not modified. But to truly integrate the DES model with the System Dynamics model or, possibly, the agent-based model, we need a separate entity to represent a modifiable variable. The DES model may perform some side effect during the simulation and update some variable. This variable can be then used in the System Dynamics model, or even another DES model.

Therefore Aivika provides with the `Var` class type:

```
type Var<'a> =
    new: e:Env * i:Dynamics<'a> -> Var<'a>
    new: e:Env * i:'a -> Var<'a>
    member Env: Env
    member Value: Dynamics<'a>
    member SetValue: a:'a -> Dynamics<unit>
```

We use the `Value` property to get the current value. This is a computation of the `Dynamics` type. To update the variable, we pass on the new value to the `SetValue` method inside some computation. To return correct results, the variable must guarantee that its value is synchronized with the simulation state. To synchronize, the variable examines the event queue whether all pending events are processed. We pass this queue during a creation of the variable. All elements of the model that depend on this event queue will be synchronized then.

Actually, the process-oriented simulation and the agent-based model require the event queue too. They need it to functionate properly and to synchronize their state with the main simulation. The event queue takes a central place in the Aivika modeling library.

External Functions

When we read a manual about some modeling tool, we can usually find a chapter devoted to that how to use external functions within the simulation. In our case the external functions can be used everywhere. It is possible due to the fact that the `Dynamics` and `DynamicsCont` computations are monads. The corresponded F# workflows easily allow you to incorporate almost any F# code including calls of your functions into the simulation. Only some care must be taken.

If your function is pure and has no side-effect then it can be used in the simulation without any restrictions. Otherwise, you should apply one of the approaches available in the Aivika modeling library.

Common Simulation Interface

A computation of the `Dynamics` type in case of the System Dynamics model can be rather slow during simulation especially if we use the `integ` function described above. At the same time I see the `Dynamics` computation mostly as an intersystem tie between different parts of the hybrid discrete-continuous model. For example, nothing prevents us from encoding the System Dynamics model as a separate high-performance and cache-friendly simulation which can be then connected to other parts of the hybrid model through the common interface of the `Dynamics` type, although some overhead is still inevitable. I think this direction of research is perspective. It will allow us to improve the performance of simulation in Aivika significantly.

Moreover, with help of the `Dynamics` type we can connect together very different simulations generated by different software tools. For example, in case of need we can wrap a simulation generated by such tool as Simtegra MapSys. This wrapper can be represented as a computation of the `Dynamics` type, but internally it will be a completely different simulation built on other ideas. Then this simulation can be involved in the hybrid simulation through Aivika. I guess that we can do the same with Vensim's simulation as well. This is also a perspective direction of research in my opinion.

Thus, the `Dynamics` type can be a common simulation interface.

Conclusion

My method is very simple. The main idea is to represent the simulation task as a dynamic process varying in time. Thanks to the powerful capabilities provided by the F# compiler, we can simplify many routine tasks of transforming our simulation model to the dynamic process that we can simulate on the computer. We can transform the System Dynamics model, the DES and agent-based models on the same platform. It allows us to create rapidly very complex hybrid discrete-continuous simulations. There is one drawback, though. If we don't take additional steps then the simulation can be slow.

Finally, I think that the described approach is not just a detail of some simulation library. This is the whole concept based on very general idea of using the dynamic processes. Since every such process is a monad value, we can easily create and bind arbitrary processes into more complex ones. Moreover, such ability to constructing is a common trait of any monad, not the particularity of the dynamic processes. However, it is important for us that this approach gives us a universal and flexible constructor to build the simulation models and the Aivika modeling library is a working proof-of-concept.