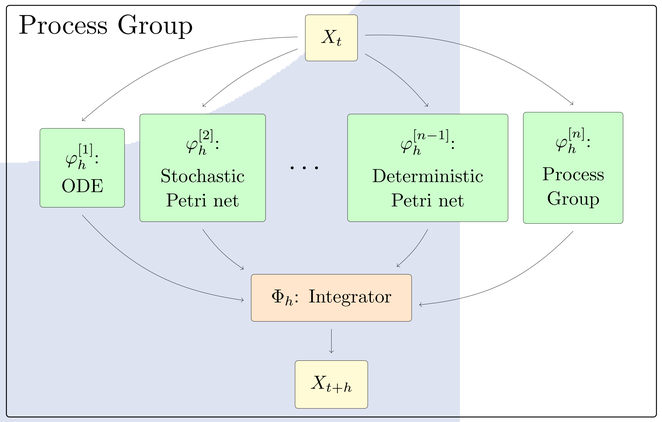
Module Integration Simulator

Dominik Bucher, Ricardo Honorato-Zimmer, William Waites

Abstract: Much work in simulating biological and other complex systems manifests in throw-away code. We provide a framework [1] for modelling these systems in a modular way, facilitating recombination and reuse of their consituent parts. The framework is incorporated into the Scala programming language to benefit from strongly typed expressions. Strong typing eliminates an whole class of potential programming errors and helps ensure a well-defined boundary between heterogeneous modules or processes. Simultaneously we provide embedded domain-specific languages for clearly and succinctly describing the mathematical behaviour of these processes. Efficient and accurate ways of integrating the flows from these processes into a complete model is problem-specific so we provide a way to flexibly compose different integration techniques.

Model

A model encapsulates a numerical experiment. It consists of state variables, X_t, and processes that act on these to produce a new state at some time in the future and integrators that combine the effects of the processes. The variables are typed and built on a strong foundation and efficient implementation of mathematical entities, such as rings, fields and vector spaces [2].



Each of the major entities, variables and processes, carry *annotations*. The annotations create an Entity-Attribute-Value structure that is used for several purposes – unification of variables across processes, resolution and translation units and production of descriptive metadata about the model and its constituent parts.

Using annotations and the clear boundary between processes facilitates collaborative development of models. *Citizen modelling* becomes possible as processes are independently developed and recombined.

Process

A process takes a (subset of) the model state at some time, t and evolves it to some time $t + h$ in the future. It is thought of as a *flow*, $\varphi_h(X_t)$. There is no *a priori* restriction on the nature of the process or the kinds of changes that it can make and modellers are free to make arbitrary kinds of processes simply by implementing a step method:

```
class MyProcess extends Process {
  override def step(t: Double, h: Double) {
    super.step(t, h)
    // implementation
  }
}
```

We have some common kinds of process availabe for use:

- Ordinary Differential Equation
- Recurrence Relation or Discrete Process
- Stochastic Differential Equation (MCMC)

Several flavours of front-ends are provided to enhance these with natural syntax for describing the processes in a given domain,

- Reaction Networks or Petri Nets

```
reactions( A + 2(B) --> C + D at 1.0 )
```

- Classical Hamiltonian Systems

```
H(q)(p) := pow(p,2)/2 + g*(1-cos(q))
```

Integrator

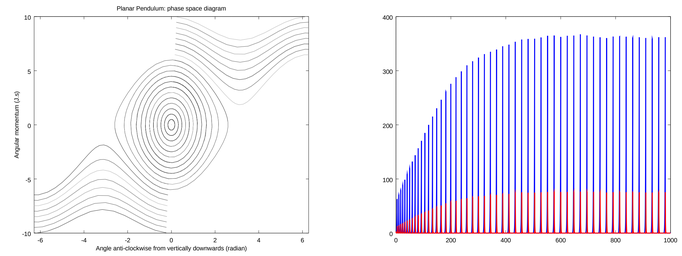
Integrating heterogeneous coupled processes is a subject of active research in numerical analysis [3]. A simple way is just to compose all of the individual processes one after another,

$$X_{t+h} \approx \Phi_h \circ X_t = \varphi_h^{[n]} \circ \varphi_h^{[n-1]} \circ \cdots \circ \varphi_h^{[2]} \circ \varphi_h^{[1]} \circ X_t$$

this is implemented as NaiveIntegrator which also shuffles the ordering at each timestep. This approach doesn't lend itself well to parallelism since each composition needs the result from the previous one. We can do better if we have a way to estimate the drift that results from artificially de-coupling the processes.

We have a variable timestep composition method [4], WeisseIntegrator that checks that the total relative change introduced is less than a certain threshold value and adjusts the step size downwards accordingly, or upwards if it is safe to do so. As well as a KickIntegrator method which allows each process to evolve independently and then corrects for drift according to the rates of change in the spatial components.

Processes written to model a particular behaviour should be written to be understandable and maintainable by a human. The splitting that comes from this is unlikely to be optimal for integration. We are working on *dynamic splitting* and remodularisation to address this.



Generally, implementing a new integrator or composition method means writing a class that implements Integrator as follows,

```
class MyIntegrator extends Integrator {
  def apply(t: Double, h: Double, g: ProcessGroup) =
{
    // implementation
    (t_next, h_next) // return new time and timestep
}
```

Work is ongoing in a few promising directions, noteably, *speculative integration* of cheap or near uncoupled processes and compile-time *retargetting* for different architectures and GPU acceleration.

References

[1] D. Bucher, R. Honorato-Zimmer and W. Waites. (2014). Module integration simulator, GitHub, [Online]. Available: `https://github.com/edinburgh-rbm/mois` (visited on 18/10/2014).

[2] T. Switzer and E. Osheim. (2014). Spire - powerful new number types and numeric abstractions for scala., GitHub, [Online]. Available: `https://github.com/non/spire` (visited on 17/09/2014).

[3] E. Hairer, C. Lubich and G. Wanner, *Geometric Numerical Integration - Structure-Preserving Algorithms for Ordinary Differential Equations*. 2006.

[4] A. Weisse, *Personal correspondence*, 2014.