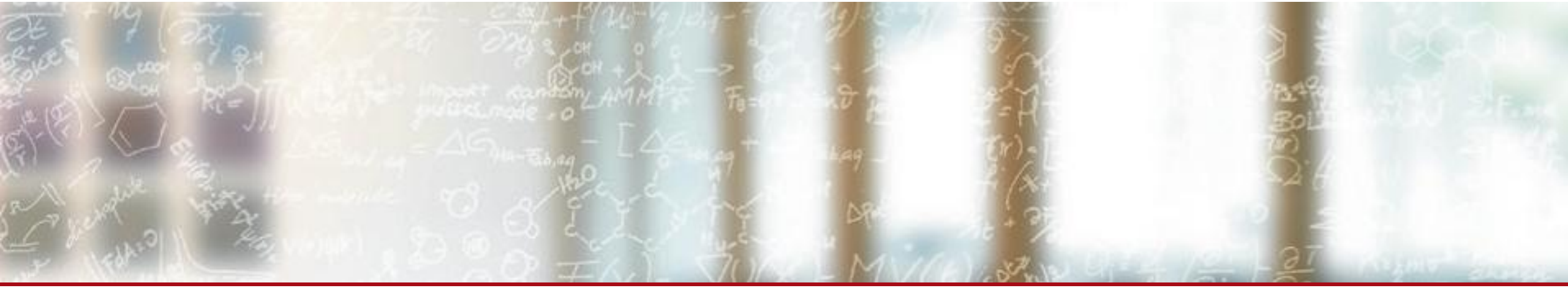


Arblang

Motivation

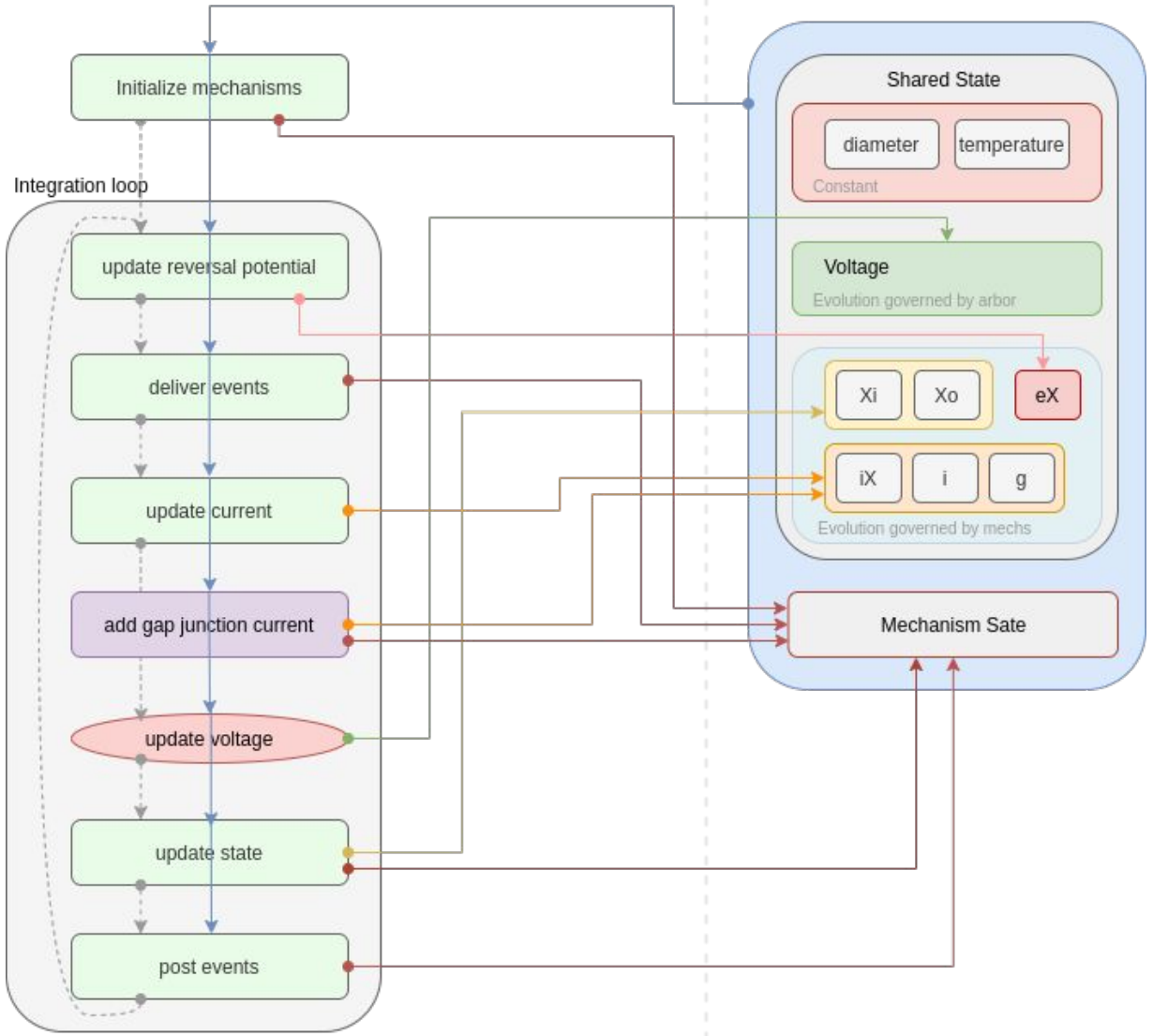
- Our current DSL, **NMODL,** is underspecified and often abused by NEURON and its users.
- **modcc**, as a result, requires continuous fixes and changes.
- Other existing DSLs are either not suitable for multicompartmental neurons, or have their own problems (NeuroML and LEMS) and they are still not widely used by the community.
- We have a chance to design our own language: **Arblang**.

Arblang - The design process

- The design process of Arblang can be split into two almost independent tasks:
  a. Designing the **language**.
  b. Designing the **compiler**.

- We can follow a top-down approach and start with designing the language.
- It is a prerequisite for designing the compiler and will guide the decisions we have to make there.
- It will help us avoid recreating some of the problems we have with **modcc**.

The integration pipeline



- This diagram is a simplification of the integration loop as it relates to mechanism callbacks.
  - LHS: the integration loop. Green blocks are programmable; red blocks are not; purple blocks will be in the future.
  - RHS: the resources available to the integration loop (the shared and mechanism states).
  - The directions of the arrows indicate the relationship between the blocks. An arrow originating from block “1” on the RHS to block “A” on the LHS indicates that block “1” is an input of block “A”. An arrow originating from block “A” on the LHS to block “1” on the RHS indicates that block “1” is an output of block “A”.
  - The main blue arrow originating from the RHS to the LHS indicates that all the mechanism callbacks have read access to all of the shared and mechanism states.

- The input/output relationships will help determine the allowed signatures of each of the mechanism callbacks.

A language proposal (1)

- The language should be **extendable** to allow new programmable blocks in the integration pipeline, or new inputs/outputs.
- The **input/output relationships** of mechanism callbacks are constrained: they should be **enforced** in the language.
- The **interface to arbor is restricted** to a few simulation parameters and the mechanism callbacks: simulator parameters should be explicitly used in the arguments and return values of the callbacks.
- The language should be **usable in other contexts** than typical mechanism models.

A language proposal (2)

- The integration pipeline is most easily translated into an **procedural** language: `init`, `update_state`, and `update_current` are called in a loop and they advance the state of simulator based on its previous values.

- But it is also useful to take a step back and think about mechanisms in a **mathematical** way. A mechanism with no ions and one state variable `m` should be capable of being described using the following formulas.

$$\begin{cases} \frac{dm(t)}{dt} = f(v(t), m(t)) \\ i(t) = g(v(t), m(t)) \end{cases}$$

- With that in mind, I present 2 language proposals with **procedural** and **symbolic** flavors. The procedural form is more flexible (not restricted to the “expected” ODE-based mechanism description), but the symbolic form offers more targeted tools.

Example: lh.mod - procedural flavor

```
// Variables in 'parameters' and 'constants' are global variables.
// Block structure used to help keep files organized.

// Read only - may be overridden by simulator
parameters {
  real gbar = 0.00001, // typed fields
  real ehcn = -45
}

// Read only and constant - can not be read or written by simulator.
constants {
  real pi = 3.14159
}

// Struct containing main mechanism state.
struct mech_state {
  real m  // typed fields
}
```

OR

```
// An alternative to `parameters` and `constants`: Global values outside of any scope.

mutable real gbar = 0.00001 // Can be overridden by simulator.
mutable real ehcn = -45     // Can be overridden by simulator.
const real pi = 3.14159     // Can not be overridden by simulator.

// Struct containing main mechanism state.
struct mech_state {
  real m  // typed fields
}
```

```
// Functional style (inspired by Scala / OCAML)
// Typed arguments and return values.
// Functions can return single values, structs or tuples.
def vtrap(real x, real y) -> real {
  y*exprelr(x/y)
}
def malpha(real v) -> real {
  0.001*6.43*vtrap(v+154.9, 11.9)
}
def mbeta(real v) -> real {
  0.001*193*exp(v/33.1)
}
```

```
// This is a pure function, NOT an arbor interfaces
def init(real v) -> mech_state {
  let t = malpha(v)
  let m = t/(t + mbeta(v))

  // Structs can be constructed from tuples of matching fields
  // or single values in the case of a struct with one field.
  mech_state(m)
}
```

```
// These are pure functions, NOT arbor interfaces
def state(real v, mech_state in) -> mech_state {
  let a = malpha(v)
  let b = mbeta(v)

  // ODEs are written in terms of `symbols`
  let m  = sym()
  let ode_m = (m' == a - m * (a + b))

  // solve takes as arguments an ODE, and a tuple binding the used
  // symbol to its initial value. It returns a tuple of the new
  // value of the symbol.
  // The returned tuple is used to create the new mech_state.
  mech_state(solve(ode_m, (m, in.m)))
}

def current(real v, mech_state in) -> (real, real) {
  let g = in.m * gbar
  let i = g * (v - ehcn)
  (i, g)
}
```

```
// The following functions represent the interface to the simulator. They bind arguments and outputs to simulator variables.
// Functions don't have or need access to all variables by default. Inputs and outputs are selected via 'binding'.

// Bindings starting with `SIM` bind to simulator state. Currently all simulator state is exposed as `real` variables.
// Bindings starting with `MECH` bind to mechanism state.Currently only MECH_STATE is allowed and it binds to the user-defined
// struct containing all the mechanisms state variables.

// The compiler can check the validity of the signature of the following functions by checking read/write permissions.
// The names of these functions can either be predefined, or selected by the user and indicated to the compiler.

defg initialize(real v: SIM_VOLTAGE) -> mech_state: MECH_STATE {
  init(v)
}

defg update_state(real v: SIM_VOLTAGE, mech_state in: MECH_STATE) -> mech_state: MECH_STATE {
  state(v, in)
}

defg update_current(real v: SIM_VOLTAGE, mech_state in: MECH_STATE) -> (real: SIM_NONSPECIFIC_CURRENT, real: SIM_CONDUCTANCE) {
  current(v, in)
}
```

Var

- var is a *varying quantity*: a function of some other variable, such as time.
- var is differentiable.
- The derivative of var: var' does not exist by itself, it is *part of the definition* of var. So, a function cannot have a var' as an argument or a return value, only var, which may or may not have a defined derivative.
- Given a var, the value of its derivative can be defined using an ODE or kinetic reaction.
- An object of type var is well-defined if it is a function of other var objects, or if its derivative is well defined, and its initial value is known.
- The full semantics are not yet clear:
  - How is the initial value defined?
  - Should var be a struct with initial value and derivative?
  - Should we merge init and state?

```
def foo () -> var {
   let x = var()
   x' = x/2 // x' is assigned
   x // has a defined derivative
}
```

```
def foo () -> (var, var) {
   let x = var()
   let y = var()
   x <-> y (1, 2)
   (x, y)
}
```

```
def foo (var a) -> var {
   a/3
}
```

Example: lh.mod - symbolic flavor

```
// parameters and constants still have the type "real"
parameters {
  real gbar = 0.00001, // typed fields
  real ehcn = -45
}

constants {
  real pi = 3.14159
}

// `var` is a varying quantity: a function of some other
// variable, such as time. It is differentiable. An object
// of type `var` is either defined as a function of other
// `var` objects, or as an initial value and a rate of change.
struct mech_state {
  var m
}
```

OR

```
// An alternative to `parameters` and `constants`: Global values
outside of any scope.
mutable real gbar = 0.00001
mutable real ehcn = -45
const real pi = 3.14159

struct mech_state {
  var m  // m has type "var"; is an implicit function of time
}
```

```
// Helper functions can be in terms of "var" or "real"
// operations between "var" and "real" are well-defined
def vtrap(var x, real y) -> var {
  y*exprelr(x/y)
}
def malpha(var v) -> var {
  0.001*6.43*vtrap(v+154.9, 11.9)
}
def mbeta(var v) -> var {
  0.001*193*exp(v/33.1)
}
```

```
// mech_state now contains members of type "var"
// init sets up the system prior to the start of the simulation
def init(var v) -> mech_state = {
  let t = malpha(v)
  mech_state(t/(t + mbeta(v)))
}
```

```
// state returns a mech_state with well-defined derivatives
def state(var v) -> mech_state {
  let a = malpha(v)
  let b = mbeta(v)

  // The construction of mech_state is special:
  // Reactions or ODE could be used.
  // m is exposed and can be used inside the constructor.
  mech_state(m' = a - m * (a + b))
}

def current(var v, mech_state in) -> (var, var) {
  let g = in.m * gbar
  let i = g * (v - ehcn)
  (i, g)
}
```

```
defg initialize(var v: SIM_VOLTAGE) -> mech_state: MECH_STATE { init(v) }
defg update_state(var v: SIM_VOLTAGE) -> mech_state: MECH_STATE { state(v) }
defg update_current(var v: SIM_VOLTAGE, mech_state in: MECH_STATE) -> (var: SIM_NONSPECIFIC_CURRENT, var: SIM_CONDUCTANCE) {
  current(v, in)
}
```

```
// Helper functions can be in terms of "var" or "real"
// operations between "var" and "real" are well-defined
def vtrap(var x, real y) -> var {
  y*exprelr(x/y)
}
def malpha(var v) -> var {
  0.001*6.43*vtrap(v+154.9, 11.9)
}
def mbeta(var v) -> var {
  0.001*193*exp(v/33.1)
}
```

```
def current(var v, mech_state in) -> (var, var) {
  let g = in.m * gbar
  let i = g * (v - ehcn)
  (i, g)
}
```

```
// state returns a mech_state with well-defined derivatives
// and initial values
def state(var v) -> mech_state {
  let a = malpha(v)
  let b = mbeta(v)
  let t = a/(a+b)
  // The construction of mech_state is special:
  // Reactions or ODEs could be used. m, a and b
  // m is exposed and can be used inside the constructor.
  mech_state(m.init = t.init, m' = a - m * (a + b))

  /* Equivalent to:
   let m = var()
   m.init = t.init
   m' = a - m * (a + b)
   mech_state(m)
  */
}
```

```
defg update_state(var v: SIM_VOLTAGE) -> mech_state: MECH_STATE { state(v) }
defg update_current(var v: SIM_VOLTAGE, mech_state in: MECH_STATE) -> (var: SIM_NONSPECIFIC_CURRENT, var: SIM_CONDUCTANCE) {
  current(v, in)
}
```

Example: kinetic scheme - procedural flavor

NMODL

```
KINETIC state {
  LOCAL alpha, beta, gamma, delta
  alpha = f_alpha(v)
  beta  = f_beta(v)
  gamma = f_gamma(v)
  delta = f_delta(v)

  ~ s <-> h (alpha, beta)
  ~ d <-> s (gamma, delta)
}
```

Arblang

```
struct mech_state {real s, real d, real h}

def state(real v, mech_state in) -> mech_state {
  let alpha = f_alpha(v)
  let beta  = f_beta(v)
  let gamma = f_gamma(v)
  let delta = f_delta(v)

  let s = sym()
  let h = sym()
  let d = sym()

  // Kinetic reaction system defined as an array of reactions.
  let kin = [s <-> h (alpha, beta),
             d <-> s (gamma, delta)]
  // solve takes as an argument the reaction system and a tuple of the previous
  // values of s, d, and h. It returns a tuple of the new values of s, d and h,
  // in the same order they were passed to the function. The tuple is used to create
  // the new mech_state object.
  mech_state(solve(kin, ((s, in.s), (d, in.d), (h, in.h))))
}
```

Example: kinetic scheme - symbolic flavor

NMODL

```
KINETIC state {
   LOCAL alpha, beta, gamma, delta
   alpha = f_alpha(v)
   beta  = f_beta(v)
   gamma = f_gamma(v)
   delta = f_delta(v)

   ~ s <-> h (alpha, beta)
   ~ d <-> s (gamma, delta)
}
```

Arblang

```
struct mech_state {var s, var h, var d}

def state(var v) -> mech_state {
  let alpha = f_alpha(v)
  let beta  = f_beta(v)
  let gamma = f_gamma(v)
  let delta = f_delta(v)

  // The constructor of a mech_state is special: it accepts any number of statements;
  // Statements can be ODEs or reactions; the members of the struct can be used in
  // the statement definitions; as long as derivative expressions can be generated
  // for each member of the struct, the constructor is considered valid.
  mech_state(s <-> h (alpha, beta),
             d <-> s (gamma, delta))
}
```

Example: kinetic scheme - symbolic flavor

NMODL

```
KINETIC state {
   LOCAL alpha, beta, gamma, delta
   alpha = f_alpha(v)
   beta  = f_beta(v)
   gamma = f_gamma(v)
   delta = f_delta(v)

   ~ s <-> h (alpha, beta)
   ~ d <-> s (gamma, delta)
}
```

Arblang

```
struct mech_state {var s, var h, var d}

def state(var v) -> mech_state {
  let alpha = f_alpha(v)
  let beta  = f_beta(v)
  let gamma = f_gamma(v)
  let delta = f_delta(v)

  mech_state(s.init = ...,
             h.init = ...,
             d.init = ...,
             s <-> h (alpha, beta),
             d <-> s (gamma, delta))

  /* Equivalent to:
   let s = var()\ let d = var()\ let h = var()
   s.init = ... \ d.init = ... \ h.init = ...
   s <-> h (alpha, beta) \ d <-> s (gamma, delta)
   mech_state(s, h, d)
  */
}
```

Language syntax - procedural & symbolic

- Functions:
  - Defined using `def`.
  - Need type annotations for arguments and results.
  - Return the last expression.
  - Cannot have argument or return type bindings to the simulator.

```
def foo (type x, type y) -> type {
   z
}
```

- Interface or "glue" functions:
  - Defined using `defg`.
  - Need type annotations *and* bindings to the simulator for every argument and for the return values.
  - Return the last expression.
  - Either have predefined names, or the user decides on the names and indicates them to the compiler.

```
defg update_state(
  real v: SIM_VOLTAGE,
  mech_state in: MECH_STATE) ->
  mech_state: MECH_STATE
{
  foo(in, v)
}
```

- Conditionals:
  - Return a result bound to a variable.
  - “else” is mandatory.
  - Braces are optional.

```
let x = if p {a} else {b}
```

- Variable bindings:
  - Scope is the next statement.

```
let a = v*v
let g = a/p1 + a/p2
3*g
```

  - Can lead to complex statements.

```
let g = let a = v*v a/p1 + a/p2 g*3
```

```
let g = (let a = v*v {a/p1 + a/p2})
g*3
```

- Tuples:
  - Used to initialize structs with compatible fields.
  - Used as arguments or return values of functions.
  - Callback functions returning multiple values, should return them in the form of tuples.
  - To destructure: bind to multiple variables.

```
struct foo { real x,real y,real z }
bar = foo(1, 2, 3)
let a, b, c = bar
```

  - Can be used to bind symbols to previous values in ODEs/kinetic reactions.

```
kin = ...
// Nested tuple input, tuple output
let a, b = solve(kin, ((v0, 0), (v1, 1)))
```

Language syntax - procedural

- Structs:
  - Tuples with named members.
  - New objects can be created from tuples with the same fields.
  - Members accessible by name.

```
struct foo { type x, type y }
let bar = foo(1, 2)
let sum = bar.x + bar.y
```

- Symbols:
  - Used in ODEs and kinetic reactions.
  - Need to be bound to previous values when solving ODEs or kinetic reactions.

```
let a = sym()
```

- ODEs:
  - Integrated variables must be symbols.
  - Use == instead of =.

```
...
let m   = sym()
let ode = m' == a - m * (a + b)
```

- Kinetic Schemes:
  - List of reactions with forward and backward rates.
  - Integrated variables must be symbols.

```
let s = sym()
let h = sym()
let d = sym()
let kin = [s <-> h (alpha, 2*alpha),
           d <-> s (beta,  3*beta)]
```

- Solving ODEs and Kinetic schemes:
  - The *solve* method.
  - Arguments are the ODE/kinetic system; and a tuple of symbol to previous value bindings.
  - *solve* returns tuples corresponding to the new values of the symbols, in the order they were provided.

```
my_struct {
  real m
}
let a = ...
let b = ...
let m = sym()
let ode = (m' == a - m * (a + b)
my_struct(solve(ode, ((m, 0))))
```

```
my_struct {
  real s, real d, real m
}
...
let s = sym()
let d = sym()
let m = sym()
let sys = [s <-> d (a, b),
           m' == a-m*(a+b)]
let new_s, new_d, new_m =
  solve(sys, ((s, in.s), (d, in.d), (m, in.m)))
```

Language syntax - symbolic

- Structs:
  - Tuples with named members.
  - New objects can be created from tuples with the same fields.
  - Members accessible by name.
  - For structs with `var` members: to simplify defining `var` derivatives, a struct object can be constructed from one or more ODEs or kinetic reactions.

```
struct foo { var x, var y, var z}
let bar = foo(1, 2)
let sum = bar.x + bar.y
foo(z' = 2*z, x <-> y (1, 2))
// The system of ODEs is constructed
// and the derivatives of each `var`
// member are assigned
```

- ODEs:
  - Used to construct the derivative of a `var` typed object.

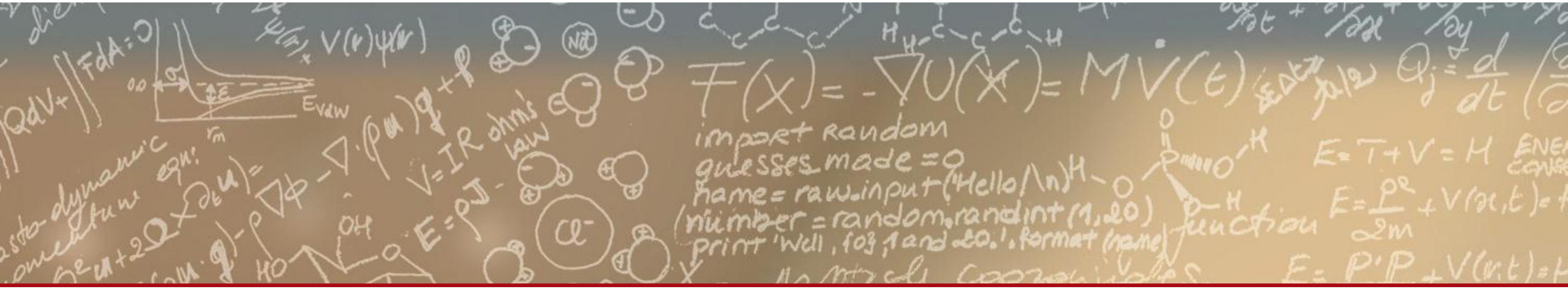
```
let m  = var()
m' = a - m * (a + b)
// derivative of m is now defined
```

- Reactions:
  - Translated into ODEs.

```
let s = var()
let d = var()
let h = var()
s <-> h (alpha, 2*alpha)
d <-> s (beta,  3*beta)
```

Bindings

- Input to all mechanism methods:
  - SIM_VOLTAGE
  - SIM_TEMPERATURE
  - SIM_DIAMETER
  - SIM_ION_CURRENT("ion_name")
  - SIM_ION_INT_CONC("ion_name")
  - SIM_ION_EXT_CONC("ion_name")
  - SIM_ION_REV_POT("ion_name")
  - MECH_STATE
- Input to method receiving events:
  - SIM_WEIGHT
- Input to method processing self-events:
  - SIM_TIME_SINCE_SPIKE
- Output of method updating current:
  - SIM_NONSPECIFIC_CURRENT
  - SIM_CONDUCTANCE
  - SIM_ION_CURRENT("ion_name")
- Output of method initializing/updating state:
  - SIM_ION_INT_CONC("ion_name")
  - SIM_ION_EXT_CONC("ion_name")
  - MECH_STATE
- Output of methods receiving events/processing self-events:
  - MECH_STATE


F(x) = -∇U(x) = MV(t)
import random
guesses_made = 0
name = raw_input('Hello! \n)H
(number = random.randint(1,20)
Print 'Well, {0}, I and 20.', format(name)
E = T + V = H
E = P'P + V(r,t) = H

Questions?

Example: expsyn.mod - procedural flavor

```
parameters {
  real tau = 2.0,
  real e = 0
}
struct mech_state {
  real g
}
def init() -> mech_state {
  mech_state(0)
}
def state(mech_state in) -> mech_state {
  let g  = sym()
  let ode_g = (g' == g/tau)
  mech_state(solve(ode_g, (g, in.g)))
}
def current(real v, mech_state in) -> real {
  in.g * (v - e)
}
def event(real weight, mech_state in) -> mech_state {
  in.g + weight
}
```

```
// Mechanism callbacks
defg initialize() -> mech_state: MECH_STATE {
  init()
}

defg update_state(mech_state in: MECH_STATE) -> mech_state: MECH_STATE {
  state(v, in)
}

// If the current update doesn't return a conductance value,
// it should be inferred by the compiler.
defg update_current(real v: SIM_VOLTAGE, mech_state in: MECH_STATE) ->
  real: SIM_NONSPECIFIC_CURRENT {
  current(v, in)
}

defg receive_event(real weight: SIM_WEIGHT, mech_state in: MECH_STATE) ->
  mech_state: MECH_STATE {
  event(weight, in)
}
```

Example: expsyn.mod - symbolic flavor

```
parameters {
  real tau = 2.0,
  real e = 0
}
struct mech_state {
  var g
}
def init() -> mech_state {
  mech_state(0)
}
def state() -> mech_state {
  mech_state(g' = g/tau)
}
def current(var v, mech_state in) -> var {
  in.g * (v - e)
}
def event(real weight, mech_state in) -> mech_state {
  in.g + weight
}
```

```
// Mechanism callbacks
defg initialize() -> mech_state: MECH_STATE {
  init()
}

defg point_state() -> mech_state: MECH_STATE {
  state(in)
}
defg point_current(var v: SIM_VOLTAGE, mech_state in: MECH_STATE) ->
  real: SIM_NONSPECIFIC_CURRENT {
  current(v, in)
}
defg receive_event(real weight: SIM_WEIGHT, mech_state in: MECH_STATE) ->
  mech_state: MECH_STATE {
  event(weight, in)
}
```

Example: expsyn.mod - symbolic flavor

```
parameters {
  real tau = 2.0,
  real e = 0
}
struct mech_state {
  var g
}
def state() -> mech_state {
  mech_state(g.init = 0, g' = g/tau)
}
def current(var v, mech_state in) -> var {
  in.g * (v - e)
}
def event(real weight, mech_state in) -> mech_state {
  in.g + weight
}
```

```
defg point_state() -> mech_state: MECH_STATE {
  state(in)
}

defg point_current(var v: SIM_VOLTAGE, mech_state in: MECH_STATE) ->
  real: SIM_NONSPECIFIC_CURRENT {
  current(v, in)
}

defg receive_event(real weight: SIM_WEIGHT, mech_state in: MECH_STATE) ->
  mech_state: MECH_STATE {
  event(weight, in)
}
```

Example: CaDynamics.mod - procedural flavor

```
// Global variables - can be overridden by simulator
parameters {
  real F       = 96485.3321233100184,
  real gamma   = 0.05,
  real decay   = 80,
  real depth   = 0.1,
  real minCai  = 1e-4,
  real initCai = 5e-5
}

// This is not the mechanism state, because the contained
// variable is ultimately owned by the simulator.
// A struct is not needed, but is used for illustration.
struct ion_state {
  real cai
}

def init() -> ion_state {
  ion_state(initCai)
}
```

```
// This function returns a struct. However, the tuple returned by `solve`
could have been returned instead.
def state(real ica, ion_state s) -> ion_state {
  let cai = sym()
  let ode = (cai` == -5000*ica*gamma/(F*depth) - (cai - minCai)/decay)
  ion_state(solve(ode, (cai, s.cai)))
}

// `init` returns a struct. But `mech_init` needs to return a real value
// to bind to a simulator variable
defg mech_init() -> real : SIM_ION_INT_CONC("ca") {
  init().cai
}

// Similarly, state needs a struct argument and returns a struct.
// `update_state` needs real values to interface with the simulator.
defg update_state(real ica: SIM_ION_CURRENT("ca"), real cai:
  SIM_ION_INT_CONC("ca")) -> real : SIM_ION_INT_CONC("ca")
{
  state(ica, ion_state(s)).cai
}
```

Example: CaDynamics.mod - symbolic flavor

```
// Global variables - can be overridden by simulator
parameters {
  real F       = 96485.3321233100184,
  real gamma   = 0.05,
  real decay   = 80,
  real depth   = 0.1,
  real minCai  = 1e-4,
  real initCai = 5e-5
}

struct ion_state {
  var cai
}

def init() -> ion_state {
  ion_state(initCai)
}
```

```
def state(var ica) -> ion_state {
  ion_state(cai' = -5000*ica*gamma/(F*depth) - (cai - minCai)/decay)
}

defg mech_init() -> var : SIM_ION_INT_CONC("ca") {
  init().cai
}

defg update_state(var ica: SIM_ION_CURRENT("ca")) ->
   var : SIM_ION_INT_CONC("ca") {
  state(ica).cai
}
```

Example: CaDynamics.mod - symbolic flavor

```
// Global variables - can be overridden by simulator
parameters {
  real F       = 96485.3321233100184,
  real gamma   = 0.05,
  real decay   = 80,
  real depth   = 0.1,
  real minCai  = 1e-4,
  real initCai = 5e-5
}

struct ion_state {
  var cai
}
```

```
def state(var ica) -> ion_state {
  let t = -5000*ica*gamma/(F*depth)
  ion_state(cai.init = initCai,
            cai' = t - (cai - minCai)/decay)
}

defg update_state(var ica: SIM_ION_CURRENT("ca")) ->
   var : SIM_ION_INT_CONC("ca") {
  state(ica).cai
}
```