ModML

A model representation language derived from an existing functional programming language

The status quo

- Two approaches taken by modelling languages:
  - aim at a particular domain (e.g. SBML). Everyone uses a different language for their own domain – making multiscale modelling harder. Innovation in modelling is slower because the language has to catch up.
  - aim to represent a particular a mathematical construct generally (e.g. CellML models usually represent DAEs). Models are in terms of DAEs, and metadata attempts to give them meaning. This gives large file sizes, duplicated information, and the risk of out-of-date metadata.

Procedural vs declarative

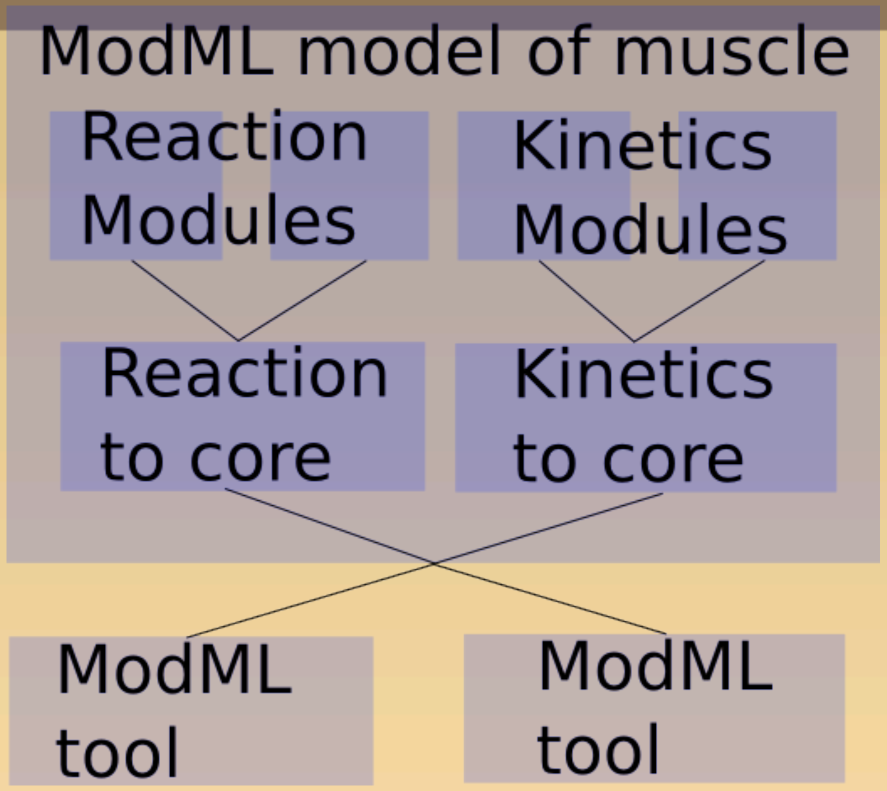
- Modellers have traditionally used procedural languages like FORTRAN and MATLAB.
  - More powerful – Turing complete, but...
  - Conflation of numerical algorithms with the models.
  - Can usually only do one thing with the model.
  - Poor composability.
- CellML and SBML are declarative – avoid the above problems but are less powerful.

Functional languages

- Pure functional programming languages provide a compromise between power and usability.
- Side-effects that might stop composition are avoided – each function computes a value based solely on the input parameters, with no hidden state.
- Functional programming can be used to express a datastructure declaring the model structure.

Generic language, domain specific modelling

- The increased power of functional programming languages allows modules, within models, to convert domain specific modules into generic terms.



ModML core

Example of ModML core code:

```
mymodelBuilder = do
  x <- newRealVariable
  y <- newRealVariable
  initialValue {- at time -} 0 {-, -} x {- = -} 10
  initialValue {- at time -} 0 {-, -} (derivative x) {- = -} 10
  (derivative (derivative x)) `newEq` (realConstant 10)
  initialValue 0 y (-5)
  initialValue 0 (derivative y) 0
  (y .+. x) `newEq` (realConstant 5)
```

- initialValue is a short-hand for an equation that equates an expression to a real value at a particular time.
- This code uses a monad which produces the model through a series of steps.
- This approach allows composition simply by combining monadic expressions:

```
combinedModel = do
   modelToBeComposed1
   modelToBeComposed2
```

ModML units

- ModML core doesn't include physical units support, but this can be built on top of ModML core by a simple transformation.
- The support for units is part of the model, not the tool – it describes how to translate units-supporting ModML into ModML core.
- Units checking occurs during the translation to plain ModML.

Biochemical reaction networks

- Support for reaction networks is not built in to ModML core.
- Instead, a function translates from a reaction datastructure into ModML core. This function is part of the model, but is re-used in many models.
- This module could intelligently link up references to the same species in the same physical compartment, allowing for better composability.

More advanced functionality

- Models could include information about chemical structure and energetics of species and reactions.
- Automated checking for conservation of charge, mass, energy, or particular atoms.

Multi-domain modelling

- Because the core isn't specific to any problem domain like biochemical reaction networks, natural support for new domains can be added.
- Models can cross multiple domains.
- Software for processing models doesn't need to know about domains – only ModML core.

Metadata

- ModML core allows for arbitary metadata to be attached to variables or equations.
- Model transformations can generate metadata at the same time as they generate the DAEs – so there is no duplication as in other languages.
- Expressing data only once ensures the two forms don't become out of syncronisation.