Functional Curation: Potential Future Directions for SED-ML

Jonathan Cooper Gary Mirams

Computational Biology Group
Department of Computer Science
University of Oxford

3rd September 2011

Outline

Functional Curation

- How can we compare models?
- Which model is best suited to investigating this experiment?
- What functionality does a model have?

We are implementing a system to answer these questions.

Using protocols

- A protocol is the *in silico* version of an experiment, that can be applied to models of the system in question

Using protocols

- A protocol is the *in silico* version of an experiment, that can be applied to models of the system in question
- A language is needed to describe protocols
  - SED-ML?

Using protocols

- A protocol is the *in silico* version of an experiment, that can be applied to models of the system in question
- A language is needed to describe protocols
  - SED-ML?
- In SED-ML, a protocol specifies the model

Using protocols

- A protocol is the *in silico* version of an experiment, that can be applied to models of the system in question
- A language is needed to describe protocols
  - SED-ML?
- In SED-ML, a protocol specifies the model
- We want one protocol, many models

Using protocols

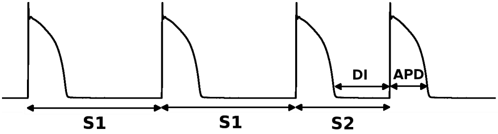
- A protocol is the *in silico* version of an experiment, that can be applied to models of the system in question
- A language is needed to describe protocols
  - SED-ML?
- In SED-ML, a protocol specifies the model
- We want one protocol, many models
- One model, many protocols

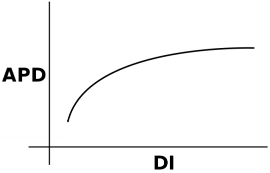
Using protocols

- A protocol is the *in silico* version of an experiment, that can be applied to models of the system in question
- A language is needed to describe protocols
    - SED-ML?
- In SED-ML, a protocol specifies the model
- We want one protocol, many models
- One model, many protocols
- Many models, many protocols

Complex post-processing

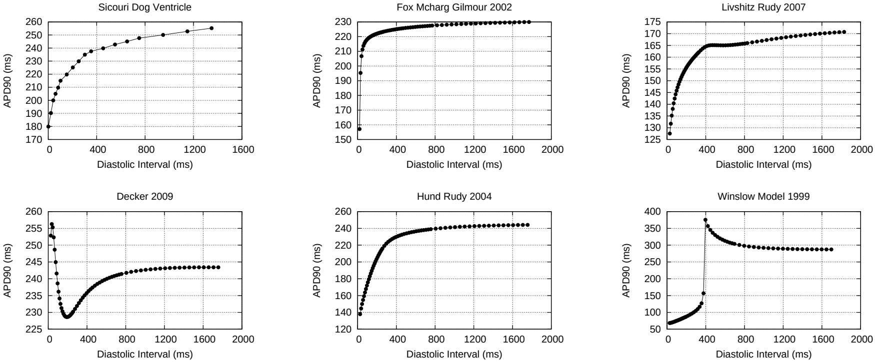
- SED-ML `dataGenerator`s are currently fairly restrictive
- Many standard cardiac protocols require additional functionality
- Example: S1-S2 restitution





Example: S1-S2 protocol on canine models

Our system currently has its own protocol language. We'd like to use SED-ML instead!



See also doi:10.1016/j.pbiomolbio.2011.06.003

Suggested extensions to SED-ML

These are features our protocol language has, that we would like to see added to SED-ML, in order to represent a wider range of experiments.

- Interfacing protocols with models
    - Ontologies, units, model inputs and outputs

Suggested extensions to SED-ML

These are features our protocol language has, that we would like to see added to SED-ML, in order to represent a wider range of experiments.

- Interfacing protocols with models
  - Ontologies, units, model inputs and outputs
- Sequenced and nested simulations
  - Based on Frank Bergmann's proposal

Suggested extensions to SED-ML

These are features our protocol language has, that we would like to see added to SED-ML, in order to represent a wider range of experiments.

- Interfacing protocols with models
  - Ontologies, units, model inputs and outputs
- Sequenced and nested simulations
  - Based on Frank Bergmann's proposal
- *N*-dimensional array based post-processing
  - Minimal extensions to MathML, many possibilities

Suggested extensions to SED-ML

These are features our protocol language has, that we would like to see added to SED-ML, in order to represent a wider range of experiments.

- Interfacing protocols with models
  - Ontologies, units, model inputs and outputs
- Sequenced and nested simulations
  - Based on Frank Bergmann's proposal
- *N*-dimensional array based post-processing
  - Minimal extensions to MathML, many possibilities
- Protocol libraries for usability
  - Allow different UIs to target different levels of user

The parts of our protocols

- Documentation
- Input specifications
- Protocol imports[1]
- Library definitions
- Units definitions
- Model interface
- Simulations
- Post-processing
- Output specifications
- Plot specifications

[1] Not yet implemented

Referring to model variables

- SED-ML uses XPath to locate variable elements
- What if models use different naming conventions or structures?
- Instead, use ontological annotation of variables
- Protocol can use `prefix:name` notation as for XML namespaces
- No need for ‘approved’ ontology — just need model & protocol to agree

Units conversions

- Different models use different units
- Protocol declares the units it uses, and conversions applied automatically
- "Biology-aware" conversion rules can be defined
  - A unary function for converting a value from one dimension to another
  - Can refer to model variables using ontology terms
  - Fall-back to next rule if required variables don't exist
  - See also doi:10.1016/j.pbiomolbio.2011.06.002

Model modifications

- Abstraction of SED-ML `Change` functionality
- A model is viewed as a system of equations, independent of modelling language

Model modifications

- Abstraction of SED-ML `Change` functionality
- A model is viewed as a system of equations, independent of modelling language
- Protocol specifies which variables are inputs and outputs
- Inputs become parameters that can be set by the protocol
  - e.g. voltage clamp experiment
- Only those equations required for the given outputs need be computed

Model modifications

- Abstraction of SED-ML `Change` functionality
- A model is viewed as a system of equations, independent of modelling language
- Protocol specifies which variables are inputs and outputs
- Inputs become parameters that can be set by the protocol
  - e.g. voltage clamp experiment
- Only those equations required for the given outputs need be computed
- Equations may also be added or replaced
  - e.g. to specify a stimulus current waveform

Sequenced and nested simulations

- An experiment may have 'setup' and 'measurement' phases $\implies$ simulations should be able to run in sequence
  - Simulations may define a prefix, allowing outputs from any simulation in a sequence to be addressed

Sequenced and nested simulations

- An experiment may have ‘setup’ and ‘measurement’ phases $\implies$ simulations should be able to run in sequence
  - Simulations may define a prefix, allowing outputs from any simulation in a sequence to be addressed
- Nesting simulations supports parameter scans, repeated runs, distribution sampling, etc.
  - Model outputs therefore become regular *n*-dimensional arrays

Modifiers

- Each simulation can have a collection of modifiers
  - There are 3 kinds of modifier:

SaveState	store the current model state, giving it a name
ResetState	reset the model to a stored state or initial conditions
SetVariable	set the value of a model variable The value is given by an expression in the post-processing language, and can access the current range value for this or any outer loop.

  - Each can be applied just at the start or end of a simulation, or prior to each loop

Post-processing language

- Aim to support complex operations with minimal implementation overhead
- Therefore base on MathML, with as few as possible added `csymbol`s
- Key features:
  - Operators for working with *n*-dimensional arrays
  - Sequencing operations (assignments to variables, assertions)
  - Defining functions (that can also be passed to other functions)
- Not just used for post-processing: also input specifications, library definitions, etc.
- Technically, this is a pure functional *n*-dimensional array based programming language

Main special expressions

`newArray` Create a new array

- by listing elements (which may be arrays)
- by comprehension using a generator expression with index ranges (abusing `domainofapplication`)

`view` Extract a sub-array

- Can use arbitrary (even negative) strides over any dimension, with wildcards

`map` Map an *n*-ary function onto *n* arrays element-wise

`fold` Collapse an array along a single dimension using a binary function

- Used to define `sum`, `max`, etc.

`find` Find indices where the operand array is non-zero

`index` Create a sub-array containing only the given indices

- Various options for avoiding irregular results

Summary of proposals

- Interfacing protocols with models
  - Refer to model variables with ontology terms
  - Apply units conversions, with user-defined rules
  - Modify model equations if required/possible
- Sequenced and nested simulations
  - Hence outputs are *n*-dimensional arrays
  - Vector ranges using post-processing language to compute 1d array
  - Modifers: save/load model state, set variable
- Array-based post-processing
  - Functions can be defined in the language
  - Operations can be sequenced
  - Array comprehensions, views, map, fold, find & index
- Protocol libraries for usability

Extra slides

The slides that follow are not central to the talk, but have extra bits of information that might be useful for questions.

Ranges

- Each simulation requires a range over which to iterate for generating output points
  - `UniformRange`, `VectorRange` and `FunctionalRange` have been proposed
  - Using post-processing language constructs to define an array of values, our `VectorRange` can implement all of these

Nested protocols

(Not yet implemented)

- Since a protocol has inputs and outputs, it can be viewed as a kind of model
- The “system of equations” abstraction does not apply, so model modifications are not possible
- This does effectively allow us to interleave post-processing and simulation however
- So we can do e.g. dynamic restitution without breaking the ‘regular *n*-d array’ data model

Environments

- A store mapping names to values
- Bindings may not be overwritten
- Multiple environments exist
  - e.g. for protocol inputs, library, model variables, simulation ranges & outputs, post-processing operations & results, function locals
- Environments can delegate lookups
  - Prefixed references go to the associated environment (e.g. specific simulation, imported protocol, model variable)
  - Many also have a default delegatee

Statements

- The `statementList` is used for function bodies, and the library & post-processing sections
- 3 kinds of statement:
  - Assignment: MathML `eq`
  - Return: `return` — only valid in functions
  - Assert: `assert` — for checking arguments etc.

Miscellaneous technicalities

- Environments binding names to (immutable) values
- Accessors (IS_ARRAY, SHAPE, etc.)
- Tuples
- Default parameters
- Wrapping MathML operators as functions
- Location information for user-friendly error messages