We have, in the last few years, witnessed the development and availability of an ever increasing number of computer models that describe complex biological structures and processes. The multi-scale and multi-physics nature of these models makes their development particularly challenging, not only from a biological or biophysical viewpoint but also from a mathematical and computational perspective. In addition, the issue of sharing and reusing such models has proved to be particularly problematic, with the published models often lacking information that is required to accurately reproduce the published results.

The International Union of Physiological Sciences Physiome Project was launched in 1997 with the aim of tackling the aforementioned issues by providing a framework for the modelling of the human body. As part of this initiative, the specifications of the CellML mark-up language were released in 2001.

Now, more than 7 years later, the time has come to assess the situation, in particular with regard to the tools and techniques that are now available to the modelling community. Thus, after introducing CellML, we review and discuss existing editors, validators, online repository, code generators and simulation environments, as well as the CellML Application Program Interface. We also address possible future directions including the need for additional mark-up languages.

**Keywords:** IUPS Physiome Project; mark-up language; CellML; XML; API; modelling environment
1. Introduction

Over the last few years, the development of new experimental tools, methods and technologies has generated large amounts of data and contributed to our improved understanding of human anatomy and physiology. Closely related to the rapid growth of biological data is the development of computers, computer science, applied mathematics and the new field of computational modelling. The availability of large amounts of biological data, powerful computers and numerical methods drive today’s development and the use of complex computational and physiological models. Computer models have become valuable tools for the understanding of multi-scale and multi-physics phenomena that underlie complex biophysical structures and processes. *In silico* models allow information acquired from different physical scales and experiments to be combined and to complement each other, providing a better picture of the involved processes and structures. Not surprisingly, the high complexity of biophysics translates into complex mathematical models, limiting the number of research centres that develop and make efficient use of such models. We can identify two main issues that contribute towards making computational physiology a challenging field.

— Computational physiology is a multidisciplinary field and may require background in biology, biophysics, mathematics and computing during the development and the use of computational models. In particular, the implementation of the mathematical models may be a time-consuming process that needs advanced numerical methods and computer programming skills.
— The tasks of model sharing and reuse are undermined by the current model publication paradigm and by the ever increasing complexity of the models. The current gold standard source for a model is the peer-reviewed publication of the model and the results described therein. From the published set of equations, other scientists may attempt to reproduce the published results or to incorporate part of the model into their own new model by writing their own computer program, but this is not always an easy task as the publication may be missing some parameters and/or contain some typographical errors.

Here, we review and present the current status of a solution addressing these issues, which is based on CellML and associated tools and techniques. CellML is an open standard format for defining and exchanging biological models. It is based on the eXtensible Mark-up Language (XML; see http://www.w3.org/TR/REC-xml/), a structured document format that can be read by both humans and machines. Owing to its acceptance by the Web community as a standard, vast amounts of software that read, write, manipulate and process XML documents are both freely and commercially available. CellML has a modular structure that facilitates the description of complex interconnected cell models. Its specifications were released in August 2001 (CellML 1.0; see http://www.cellml.org/specifications/cellml_1.0/) and subsequently refined in February 2006 (CellML 1.1; see http://www.cellml.org/specifications/cellml_1.1/). Both versions offer a document-based architecture, but only CellML 1.1 allows a modeller to efficiently reuse (import) model descriptions. This import feature is useful in allowing mathematical modellers to build more complex models based on the previously published models without needing to implement the original model.
CellML is a flexible and powerful language, as demonstrated by the number (more than 330) and variety of models available in the CellML model repository (see http://www.cellml.org/models/), including electrophysiological and mechanical models, as well as biochemical pathway models. However, whereas the CellML specifications *per se* do not solve the aforementioned problems, the CellML-based tools currently available provide a new paradigm for model development. Appropriate editors, automatic code generators, graphical user interface (GUI) environments, Web portals, simulators and repositories provide a powerful and user-friendly environment for computational physiology. The complex techniques and know-how that are behind computational modelling, such as advanced numerical methods, visualization techniques and even computer code programming are all taken care of by those tools. A variety of tools exist for CellML model creation and modification (editors) as well as for CellML debugging and verification (validators). During the development of a new model, components of the previously published models can be imported from the CellML model repository. After creation and validation, the same CellML document can be processed by rendering software to generate equations suitable for publishing or by software to generate computer code (code generators) that can then be compiled and executed to perform simulations, ensuring that the executed model and the published equations are consistent. In this paradigm, the onus of model validity no longer rests with the model user, but with the model author and the software engineers writing the CellML application libraries and tools.

In the next sections, we describe each of the components that comprise this new paradigm for computational modelling. CellML’s key features are presented in §2, while §3 introduces the different tools and techniques available today for editing, validating, sharing and curating CellML models. In addition, §3d,e present and compare different tools for automatic code generation and simulation of CellML models, respectively. Possible future directions are then addressed in §4 and some conclusions are drawn in §5.

## 2. CellML

Each model encoded in CellML has both a name and, most importantly, a unique identifier. This identifier, together with the model’s Uniform Resource Locator, forms the model’s Universal Resource Identifier, which can be used to uniquely refer to a model.

A CellML document can include elements of one of the following types: *units; component; connection; group; and import*. In addition, *metadata* (i.e. data about data) can be included anywhere within a CellML document.

(a) **Units**

Every quantity used in a model must have units associated with it. This is one of CellML’s most powerful features, ensuring robustness and reusability of CellML models and components. It allows us to have a variable, say $A$, with units of millivolt and another, $B$, with units of volt, and yet be able to have $A = B$. Conversion of $B$ from volt to millivolt would, ideally, be done by CellML processing software.

*Phil. Trans. R. Soc. A* (2008)
Components

A CellML model usually consists of several components related to one another. A component is typically used to encapsulate concepts by hiding its details from other components and by providing well-defined interfaces to other components. Components may contain units definitions, variable declarations, mathematical equations and reactions. CellML users are, however, currently discouraged from using reactions as there are ongoing discussions about removing them from future CellML specifications.

Units definitions may occur at both the model and component levels. In the former case, the new units will be available to any component within the model, while units defined within a component will have their use limited to that particular component.

(i) Variables

Variables are used to name particular quantities declared in a component and must have units associated with them. They may also have an initial value, as well as define an interface to specify whether their value is used by other components or retrieved from another component (a variable with no interface is only visible in the component where it is declared).

(ii) Mathematics

Description of a model’s mathematics is achieved through MathML content mark-up (see http://www.w3.org/Math/), thus giving access to arithmetical operators (+, −, ×, ÷, etc.), relational operators (=, ≠, <, >, etc.), logical operators (AND, OR, XOR and NOT), as well as many other operators.

(c) Connections

As mentioned in §2b(i), variables may be used by other components or retrieved from another component. In both cases, this requires a connection between the component where the value of a variable is defined and the component where that value is used, so that mapping of variables between the two components can be performed.

Only one connection is allowed between two components, which means that mappings in either direction can be recorded within a single connection with the direction being determined from the interface attribute of the variables involved.

(d) Groups

Clear relationships usually exist between components. For this reason, CellML offers a grouping mechanism by which it is possible to organize components into both geometric containment and logical encapsulation hierarchies.

The former type of group is typically used by CellML processing software to provide the user with a physical representation of a model, while the latter is used to hide parts of a model.

Components that are not part of a grouping relationship are assumed to be at the same level as the ‘top’ components. Such an assumption is particularly important in the context of the interface of a variable, since it determines the type of connection that is possible, based on the logical relationship that exists between the different components.
(e) Imports

The modular approach used by CellML provides model users with the underlying structure that is required to make the reuse of units and components of different CellML models possible. The mechanism by which such reuse can actually be achieved was introduced in CellML 1.1 and is known as import. Using the import mechanism results in a local ‘copy’ of the referenced model with the specified units and components available for use in the current model.

(f) Metadata

Metadata (e.g. reference to the publication that describes the model, references to the experimental data used to derive the model, limitations and range of application of the model) can be embedded anywhere in a CellML document by using the Resource Description Framework (RDF; see http://www.w3.org/RDF/). Such information can later be used, for example, to search for specific models and components.

(g) Best practices

The previous sections offered a quick overview of the key features offered by the CellML language. For a more in-depth description of the language, we refer the reader to the CellML 1.0 and 1.1 specifications (see http://www.cellml.org/specifications/cellml_1.0/ and http://www.cellml.org/specifications/cellml_1.1/, respectively), CellML Metadata 1.0 specifications (see http://www.cellml.org/specifications/metadata/cellml_metadata_1.0/), as well as the CellML 1.1 overview by Cuellar et al. (2003) and the CellML review by Lloyd et al. (2004), while answers to frequently asked questions can be found at http://www.cellml.org/faq/.

A current limitation of the CellML specifications is the lack of information on how best to code a model in CellML, though the CellML model repository (see http://www.cellml.org/models/) may offer a good starting point. The CellML encoding of the Noble cardiac ventricular electrophysiological models (e.g. Noble et al. 1998) by Nickerson & Hunter (2006) may also be used as an example of the current best practices. Others are the online examples, which can be found at http://www.cellml.org/tutorial/electrophysiological/ and which illustrate the use of CellML for developing electrophysiological models.

Best practices have been the subject of several recent discussions on the CellML mailing list (see http://www.cellml.org/wiki/MailingList/) where topics such as the form top-level mathematical expressions should take or the use of external code have been discussed (search the list for terms such as ‘BCP top-level mathematics operator’ and ‘BCP including external code’, respectively). Independent of those discussions, model authors are strongly encouraged to appropriately and comprehensively annotate their models, as well as reuse their models or those of others as often as possible.

Reuse of models can be done using the import mechanism as discussed in §2e and for which some examples are available at http://www.cellml.org/tutorial/cellml_1.1/, while annotation of models is done using metadata (see §2f). The format of those metadata must, however, be agreed on by the community, and though there are already well-defined and accepted formats (e.g. vCard, the
electronic equivalent of business cards; see http://www.w3.org/TR/vcard-rdf),
there are also ongoing discussions on, for instance, how best to specify simulation
and graphing metadata (Nickerson et al. 2008).

3. Tools and techniques

Since the release of CellML in 2001, the availability of an increasing number of
tools and techniques dedicated to editing (both visual and textual), validation,
sharing and curation (through an online repository), generation of code (for
external use) and execution of CellML models has been witnessed. A summary
comparison of the main tools described in this review is given in table 1.

Table 1. Tools summary.

<table>
<thead>
<tr>
<th></th>
<th>AGOS\textsuperscript{a}</th>
<th>CESE\textsuperscript{b}</th>
<th>COR\textsuperscript{c}</th>
<th>JSim\textsuperscript{d}</th>
<th>PCEnv\textsuperscript{e}</th>
<th>PyCml\textsuperscript{f}</th>
<th>VCell\textsuperscript{g}</th>
</tr>
</thead>
<tbody>
<tr>
<td>CellML version</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0/1.1</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>editing</td>
<td>import</td>
<td>import</td>
<td>yes</td>
<td>import/</td>
<td>some</td>
<td>no</td>
<td>import/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>export</td>
<td></td>
<td></td>
<td>export</td>
</tr>
<tr>
<td>validation</td>
<td>some</td>
<td>some</td>
<td>yes</td>
<td>some</td>
<td>some</td>
<td>yes</td>
<td>some</td>
</tr>
<tr>
<td>units (validation/</td>
<td>no</td>
<td>no</td>
<td>validation</td>
<td>yes</td>
<td>some</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>conversion)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>code generation</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes\textsuperscript{h}</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>simulation</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>environment supported\textsuperscript{i}</td>
<td>I</td>
<td>W/L/M</td>
<td>W</td>
<td>W/L/M</td>
<td>W/L</td>
<td>I/W/L/M</td>
<td>W/L/M</td>
</tr>
</tbody>
</table>

\textsuperscript{a}API generator for ODE Solution (AGOS; see http://www.fisiocomp.ufjf.br/).
\textsuperscript{b}Cell Electrophysiology Simulation Environment (CESE; see http://cese.sourceforge.net/).
\textsuperscript{c}Cellular Open Resource (COR; see http://cor.physiol.ox.ac.uk/).
\textsuperscript{d}JSim (see http://www.physiome.org/jsim/).
\textsuperscript{e}Physiome CellML Environment (PCEnv; see http://www.cellml.org/tools/pcenv/).
\textsuperscript{f}PyCml (see https://chaste.ediamond.ox.ac.uk/cellml/).
\textsuperscript{g}Virtual Cell (VCell; see http://www.nrcam.uchc.edu/).
\textsuperscript{h}Via CCGS (see §3\textsuperscript{d}(i)).
\textsuperscript{i}I, Internet (through a Web interface); W, Microsoft Windows; L, Linux; M, Mac OS X.

electronic equivalent of business cards; see http://www.w3.org/TR/vcard-rdf),
there are also ongoing discussions on, for instance, how best to specify simulation
and graphing metadata (Nickerson et al. 2008).

3. Tools and techniques

Since the release of CellML in 2001, the availability of an increasing number of
tools and techniques dedicated to editing (both visual and textual), validation,
sharing and curation (through an online repository), generation of code (for
external use) and execution of CellML models has been witnessed. A summary
comparison of the main tools described in this review is given in table 1.

(a) Editors

There are two aspects of editing CellML models. One involves addressing the
problem of visualizing and linking CellML models to create biologically useful
entities. The other involves the actual editing at the text level. As CellML is an
XML document structure, it is verbose and can be difficult to interpret. Thus, the
Cellular Open Resource project (COR; Garny et al. 2003\textsuperscript{a}) has developed a
compact syntax to alleviate this problem.

(i) Visual display of models

Visualizing and linking CellML models can be achieved in Virtual Cell (VCell;
Loew & Schaff 2001), a Java-based modelling and simulation environment, which
can import and export a variety of formats, including CellML. It also provides
a GUI for creating biomodels and defining geometries. CellML files are loaded into an internal C-like syntax in which, because the hierarchical structure is lacking, the resultant output has lost much of the semantic meaning between equations. A more comprehensive study of quantifying and manipulating CellML documents is being undertaken at the University of Auckland. A biological viewer and an editor are being developed to support visualizing the biology and its relationship to the underlying CellML document. The basic pipeline for generating diagrams requires representing CellML models in the OWL format (see http://www.w3.org/TR/owl-xmlsyntax/). Bindings are formed from these CellML OWL models to biological ontologies (largely represented as BioPAX models; see http://www.biopax.org/release/biopax-level2-documentation.pdf) to provide biological meaning to CellML OWL entities. A visual template ontology is developed to provide mappings to a common graphical notation and bindings are formed from the biological ontologies to the visual template ontology. Using these ontological mappings, the relevant biological model of a CellML model can be represented and visualized (figure 1).

(ii) Editing tools

COR is an environment for modelling biological function with a particular focus on cardiac electrophysiological modelling. It runs under Microsoft Windows and is targeted not only at modellers, but also at experimentalists and teachers. It offers an intuitive and user-friendly interface that is fully customizable in terms of fonts, sizes, colours, etc. COR has two modes of functioning: editorial and computational (see §3e). The former mode is used to edit CellML documents that, upon opening, are converted into a concise text format (figure 2b). This allows for quick and easy editing of a model. Other features include a command viewer that can be used to graphically visualize an equation as it would appear in a publication (figure 2a), the conversion to different formats of the internal representation of a CellML document (see §3d(iii)) and a CellML validator (see §3b(ii)).

The Physiome CellML Environment (PCEnv) aims at providing a unified way to work with both CellML 1.0 and 1.1 documents. It currently provides the ability to run simulations and the initial implementation of a full editing environment. The intention is to allow the creation of new models and enable new components to be added to them, and to provide a model development environment in which unit consistency and other model validation tests (see below) and biophysical constraints can be imposed during the model building process.

(b) Validators

The ultimate aim of validation is to prevent incorrect results. Humans will always make errors; so the more errors that can be automatically detected, the more reliable the system becomes. Detecting errors early is also important, since much time will be saved if we can find errors by an analysis of the model, rather than waiting until simulation time, or even for an analysis of the simulation results. It is important to note that there are two different senses of the word ‘validation’ in the context of CellML. The sense that is probably more familiar to modellers and physiologists is to validate a mathematical model against experimental data, to determine to what extent the model matches reality (and can therefore elucidate reality). That is an important question, but it is not
the question we address here. By validation, we mean comparing a CellML model to the definition of what constitutes a CellML document as given in the specifications. This is a much easier question to answer automatically, since we can encode the specifications in a form that computers can understand, and hence compare CellML documents against. It is also an important question if CellML documents are to be shared among different groups, and read by different tools, but still assigned the same meaning.

Figure 1. Basic pipeline for generating visual representations of CellML models (courtesy of Sarala Dissanayake of the University of Auckland).

Phil. Trans. R. Soc. A (2008)
We also address the issue of validating the dimensional consistency of the model represented by a CellML document. This issue is mentioned in the specifications, but dimensional consistency is *not required* for a CellML model to be valid according to the specifications. It is obviously desirable for models to be dimensionally consistent, since if they are not then *a priori*, they cannot be an accurate representation of reality (e.g. adding a length to a volume is nonsensical).

(i) **Standard XML validation**

As CellML is an XML-based language, the natural place to start in validating a CellML document is with standard XML validation tools. These work by comparing a CellML document against a *schema* that defines what is allowable content. XML Schema (see [http://www.w3.org/XML/Schema.html](http://www.w3.org/XML/Schema.html)) and RELAX NG (see [http://www.relaxng.org/](http://www.relaxng.org/)) are two such *schema languages* that both define XML *grammars*. These specify what elements and attributes are allowed where, and what type of textual data they may contain (e.g. strings, integers, dates). Since they lack any ‘higher level’ checks, they are not complete solutions for validating CellML, but may profitably be used as components of such a solution.

XML Schema is the most well known of these schema languages, and is a W3C specification. RELAX NG is less well known, but is also a published international standard. An XML Schema for CellML 1.1 and a RELAX NG schema for CellML 1.0 are both available from [http://www.cellml.org/cellml/](http://www.cellml.org/cellml/). There are only minor technical differences between the two schemas, and so it is mainly a matter of personal preference which to use, unless CellML 1.1 support is required.

Figure 2. COR in editorial mode ([Garny et al. 2003a]) (a) Command viewer and (b) CellML editor. The equation highlighted in grey is shown in the command viewer.

\[
\alpha_m = \frac{100 \left( -V - 48 \right)}{\left( -48 \right)} e^{\frac{15}{-1}}
\]

The equation highlighted in grey is shown in the command viewer.
Editors with validation capabilities

Some CellML tools also contain validation subsystems; both COR and JSim fall into this category. The main advantage of having a validator coupled with the model editing environment is that it is easy to go from a validation error to the offending portion of the model in the editor, enabling swifter development.

Note that all CellML-based tools have some validation capabilities, simply by virtue of needing to read a CellML document and store it in memory in some fashion. The extent to which this validation checks all the rules given in the CellML specifications varies, but is generally minor, and so we do not consider it a true validation in the sense described in this section.

JSim imports CellML models into its own internal representation, rather than working with CellML directly. The transformation is based more on the use of a wide range of example models than strict adherence to the CellML specifications, and also currently only supports CellML 1.0 models. It is thus, strictly speaking, not a validator for CellML. However, it does have good support for checking units, so it is useful in this regard (see below).

COR, however, validates models against all of the rules given in the core specifications for CellML 1.0, except those for reactions and metadata which are not currently supported. COR was developed primarily for cardiac electrophysiological modelling where reactions are not used. It thus considers models with reaction elements ‘invalid’. It should be noted, however, that the use of the reaction element is currently discouraged (see §2b). COR also implements additional rules to restrict the set of valid models to those with a similar form to cardiac electrophysiological models. These restrictions are discussed in §3d(iii).

Units validation

The usefulness of dimensional analysis in checking the correctness of the mathematical models is well known. Similarly, if quantities of the same dimension, but measured in different units, are compared without a suitable conversion, then simulations of the model are unlikely to give sensible results. Such errors may seem trivial, but they are not uncommon. Scientific programs are often composed of many different components, sometimes written by different authors, using differing units, especially for multi-scale models. Units conversions at the interfaces are, thus, essential for correct operation. Units errors can also be costly (e.g. the loss of NASA’s Mars Climate Orbiter in 1999 was due to the software failing to convert between imperial and metric units; Isbell & Savage 1999). Since such errors are easily made, automatically checking for them is crucial.

As mentioned in §2a, CellML mandates the use of units in models. We are aware of four tools with support for units checking: COR; JSim; PCEnv (at connections between components); and PyCml. However, only JSim and PyCml are fully capable of automatic units conversions; PCEnv will only perform conversions at connections between components.

There are two levels to the units checking when loading a model into JSim. The first is in the transformation of the CellML model into a JSim model. This transformation will fail when incompatible units are detected outside of the MathML content. Once a CellML model has been successfully imported into a JSim model, it can then be compiled into runnable code. At this stage, incompatibilities of units within the mathematics are detected.
The units checking within JSim is very thorough and mature, having been in use for quite a while with a large user base. Moreover, JSim has always had a similar philosophy as CellML, in that every number in a model must have units, so the units checking within JSim has always included the numerical constants in equations, as required by CellML. JSim will not, however, accept CellML models including units with non-zero offsets, since contextual information allowing differentiation between absolute and relative units is not available in CellML. For example, there is no explicit differentiation between absolute and relative temperatures, so confusion can easily arise if temperatures are doubled (e.g. twice 2°C could refer to 550.3 K or a temperature difference of 4 K).

One useful feature is that JSim can also run simulations, which can help when compatible (dimensionally consistent) but incorrect units have been used. For example, if you have a variable with units of volts instead of millivolts, computations will quickly go awry once the appropriate scale factor (1000) has been added to the equations.

PyCml includes a Web-based validation suite (a command line version is also available for download) that can check the units in CellML models directly. This tool features strict checking against the rules in the CellML specifications, using the RELAX NG schema where appropriate. It also checks that each variable has a valid ‘type’, i.e. is a free variable, a state variable, a constant, a computed variable or imported from another component (this ensures that all variables are suitably defined and not being computed in two different ways). The units checking (described in detail by Cooper & McKeever 2008) does not support the whole of MathML, but new elements can easily be added on request. Optionally, warnings can be generated if automatic units conversions would be required, since not all tools support this.

As part of its validation methodology, COR implements the novel units checking algorithms developed by Cooper & McKeever (2008). Thus, any dimensional inconsistency that exists in a CellML document is reported and, in agreement with the CellML specifications, is treated as a warning. The latter allows for any model to be run, including those that are dimensionally inconsistent (this can be useful when developing or debugging a model).

(iv) Metadata

None of the validation tools mentioned above perform any checks on metadata included within a CellML document. Such a tool would, however, clearly be desirable as the use of metadata is becoming increasingly common. Support for easily adding metadata within an editor is also key.

As tools are developed to make use of the simulation and graphing metadata (see §4c), parts of the metadata contained in a CellML document start to get used, and hence validated. Similarly, as the CellML model repository develops and model curation information is added, more of the potential pool of model metadata will be validated.

(c) Online repository

After editing and validating mathematical models encoded in CellML, there remains the issue of making them available for public use, one of the primary principals of the International Union of Physiological Sciences (IUPS) Physiome Project (see http://www.physiomeproject.org/). Associated with providing
access to models is the related issue of curation of the CellML encoding of a given mathematical model, thus providing users of such models with some quantifiable level of confidence in the model they are using.

The CellML model repository (see http://www.cellml.org/models/) began as a collection of example models used to aid the initial development of the CellML language, but evolved into a collection of CellML encodings of the published models, initially focused on cardiac cellular electrophysiology and reaction pathway models, but quickly extended to other organ systems, and both lower and higher spatial scale models. Throughout this early development, the focus was on producing CellML models that faithfully encoded the mathematics described by the original publication.

With the recent move to a repository technology based on the Zope object database and the Plone content management system (see http://www.zope.org/ and http://www.plone.org/, respectively), the model repository moved away from a collection of static pages to a more dynamic and feature-rich system. The new model repository has been developed as a Plone product known as the Physiome Model Repository (PMR; see http://www.cellml.org/tools/pmr/), which provides such features as

— access control and management;
— the ability to provide dynamic and customizable views of a given CellML model;
— the ability to search and query within and throughout the models contained in the repository; and
— the ability to annotate models as being different representations of the same underlying mathematical model.

With the PMR, all registered members of the CellML portal are able to create their own model repository and control who they grant access to at both the whole repository level and on a model-by-model basis, as required by their needs. This provides a valuable community tool for the collaborative development of models, which can be kept private until the author is ready to publish the model and, hopefully, to transfer the model to the main CellML model repository.

Zope and Plone provide numerous methods for manipulating and viewing many types of documents using standard technology such as XSL transformations, RDF queries and general Python scripting. Using these methods, the source XML serialization of a CellML model can be transformed by the PMR into many different views when requested by a model repository user. The CellML model repository currently provides views of the mathematics contained in a model, a pretty-print of the XML serialized CellML model, and a listing of all metadata contained in a model with dynamic links to the corresponding object in the XML serialization of the model. By incorporating the CellML Application Program Interface (API; see §3f) and C Code Generation Service (CCGS; see §3d(i)), the PMR is also able to provide a view of the procedural code generated from the model.

As mentioned above, until recently the model repository has been developed with the goal of providing CellML models that accurately duplicate the mathematics of a published model. As clearly shown by the tool developments reviewed in this article, it is now possible to do a lot more with a CellML model than simply compare a rendering of the mathematical equations in a given model with a published paper. Thus, it is important for the CellML model
repository not only to provide access to models but also to clearly curate those models to say what a user can expect from a particular CellML encoding of a mathematical model. With the ability of the PMR to provide access to multiple versions of a given model, it is now possible to store, in the CellML model repository, a set of CellML models that are all associated with a single original published model. We can, thus, store multiple versions of the CellML encoding of a given published model and annotate them so that users of the model are able to determine which version is best for their particular use.

At the most basic level of CellML model curation, a given CellML encoding of a mathematical model can be assigned to one of the following four curation levels.

— Level 0. The model has been implemented, but has not yet been through the process of curation.
— Level 1. The model has been implemented and corrected, if necessary, to accurately represent the published model.
— Level 2. The model has been implemented and corrected, if necessary, to accurately reproduce the published results.
— Level 3. The model has been implemented and corrected, if necessary, to satisfy domain-specific biophysical constraints (e.g. conservation of mass and charge, thermodynamic constraints).

Potentially, a CellML model may be assigned multiple curation levels, but historically, at least, CellML models that accurately represent a model as it was published (level 1) will not satisfy the requirements for level 2. With the tool developments discussed elsewhere in this review, it is hoped that new models being developed will satisfy curation levels 1 and 2 with a single version of the CellML model.

Work is currently underway to incorporate these curation levels into CellML model metadata and the PMR workflows, with this information currently serialized in models as plain text descriptions. This will provide an initial framework for the curation of models contained in the CellML model repository, and interfaces will be added to the PMR to allow this information to be provided to repository users. Active discussion is also in progress to determine the details of exactly what is required for level 3 curation.

(d) Code generators

In order to simulate a CellML model, the mathematics contained within it has to be available in a form that can be identified, extracted and converted into an appropriate format for evaluation. It is possible to do this in an interpretive fashion, by having the model stored in memory in some data structure, and querying it to evaluate expressions. This adds considerable overhead to the computation however; so most simulation environments take a different approach. When asked to simulate a model, they first translate it into code that the computer can execute directly, a process akin to compiling a computer program. This approach can also be followed to enable the use of CellML models by simulation environments that do not support CellML directly, by generating code in the language the environment is written in, which can then be compiled and linked against the simulation software.
A code generator is a tool that performs this translation, either as part of a larger program (e.g. COR) or as a stand-alone application (e.g. PyCml).

All existing code generators view a model as an initial-value problem, i.e. a system of ordinary differential equations (ODEs) with initial values given for each state variable (the trivial case where all equations are algebraic is also supported). Classifications of equations as differential or algebraic, and of variables as dependent, independent, constant or computed, are not given explicitly in the CellML document. Rather, the mathematics must be analysed in order to determine them. This is because CellML is a declarative language, specifying the relationships between variables rather than a direct process for computing them.

Most code generators support just the CellML subset of MathML, and hence do not handle constructs defining integral equations. Most also require all equations to be given explicitly, with a single term on the l.h.s. Furthermore, CellML assumes that all variables are real-valued; so complex numbers, vectors, matrices and the like are not currently supported (see §4a(i)).

(i) C code generation service

CCGS (see https://svn.physiomeproject.org/svn/physiome/CellML_DOM_API/trunk/CCGS/) is an API, built on top of the CellML API (see §3f), to enable generation of C code representing a CellML model. As for the CellML API itself, it is being developed at the University of Auckland. It is very flexible in the models it can handle, being able to process almost any valid CellML 1.1 model that defines a system of ODEs or algebraic equations. Notably, CCGS does support definite integrals and implicitly defined algebraic equations, which none of the other code generators support at present. It also performs unit conversions at connections where required, although not within equations.

CCGS targets code written in the C language (although there are plans to target other languages as well). However, it should be noted that it is an API, rather than a full program targeting end users. It thus does not output a source file, but rather provides generation information about the code, as well as code ‘fragments’ that compute certain parts of the model (constants, computed constants, rates and other variables). It is thus fairly flexible, but a program must be written using the API in order to generate C code in whatever format is desired. Several programs currently use the API, notably those in the following list.

— The CellML Integration Service (CIS), which also comes with the CellML API, uses the GNU Scientific Library and CCGS to run simulations. This is also used by PCEnv for its simulation functionality.
— The CCGS has been used to generate code suitable for use with the SUNDIALS library (see http://cellml.sourceforge.net/ and http://www.llnl.gov/casc/sundials/, respectively).
— CellML2C, a small demonstration application, is provided with the CCGS.

(ii) API generator for ODE solution (AGOS)

AGOS is a tool developed at the Federal University of Juiz de Fora, Brazil, and translates CellML 1.0 into a C++ API, which can be used to simulate the model. It handles systems of first-order ODEs sharing a common independent variable,
requiring at least one differential equation. It supports all of the CellML subset of MathML, except for semantics and annotation elements. AGOS can display the units of variables, but does not perform any checking or units conversion.

The C++ API is used to set up initial conditions and parameters and to solve the ODE system. In addition, the API offers reflexive functions giving information about the model, for example the number of variables and their names. These reflexive functions allow the automatic creation of model-specific user interfaces.

The process of code generation is based on three steps: pre-processing; feature extraction; and code generation based on a grammar template. If a new structure or even a new language in the output is needed, all that is required is to modify the last step, i.e. the grammar and/or the template (Barbosa et al. 2006).

(iii) COR

COR contains both types of code generation as mentioned in the introduction to this section. When a model is run by COR, behind the scenes the model is translated directly into machine code (alternatively, it can be converted to C and compiled using either the Microsoft or the Intel C++ compiler, should either of them be available on the user’s machine), which is used to actually compute the model. Also, COR has the facility to export CellML documents to a wide range of programming languages. Since COR was primarily developed for cardiac electrophysiological modelling, it only supports CellML 1.0 models that have at least one ODE, and all ODEs must be integrated against time. In addition, user-defined relationship hierarchies are not supported and only one encapsulation group and/or one containment group may be given in the model (the author has not yet found a need for more functionality in this area).

COR can currently export CellML documents to C, C++, Delphi for Windows 32, Fortran 77, Java, MATLAB, Pascal and TeX. For each programming language, the output source code consists of the equivalent of two methods, one initializing all variables, and the other computing the ‘r.h.s.’ of the ODE system, given the current values of the state variables and time.

Modifications to the generated code must be done either by hand or by modifying the source code of COR. The latter is very easy to do (adding a new output language typically takes between 2 hours and 2 days), but the simple nature of the code output (just two methods) means that hand tweaking to fit a given simulation framework is also fairly straightforward.

COR is currently only released as an executable, although copies of the source code may be obtained on request to the author. Eventually, it will be released under an Open Source licence, although the type of this is yet to be determined.

(iv) PyCml

PyCml is also capable of code generation, although its main functions are validation (see §3b(iii)) and optimization (described in §3d(v)). Its code generation, such as COR, is targeted at cardiac electrophysiological modelling, and thus has the same limitations. Currently, it generates C++ code for a simulation framework being developed in the Computing Laboratory at the University of Oxford, as well as Haskell and Maple codes used for some as-yet experimental optimization techniques.
Optimization

Simulation of many of the models described in CellML can be computationally intensive, due to the complexity of the biological processes described. Compiler optimization of code generated from CellML models is obviously useful in tackling this problem. There are also, however, domain-specific optimizations that have been applied to hand-coded models in the past, and it is desirable to be able to apply these to models defined in CellML also.

There has been some work on the use of domain-specific optimizations for CellML, and two types of optimization have been performed. One is a type of partial evaluation (Jones et al. 1993). CCGS performs some optimization along these lines: equations are separated into those that need only to be computed once, and those that must be recomputed after every time step (because the values of variables in the equation change with time). This approach can be generalized to pre-compute sub-expressions also, and much work has been done on such techniques in the computer science community. It is important to note, at this point, that there is a balance to be struck between optimization and flexibility. If it is desired to be able to change arbitrary initial conditions and/or parameters without recompilation, then optimizations cannot assume these are constant.

Some investigation into more aggressive partial evaluation has been undertaken (Cooper et al. 2006; Cooper & McKeever 2007), and a prototype tool is available as part of PyCml. It works by producing a new CellML document from the input model, in which as much of the model as possible has been pre-computed. This new model may then be used in any CellML processing software. The tool is capable of transforming only CellML 1.0 models in which all equations are explicit. It is a command line-based tool, written in Python, and should be easily portable to most platforms.

Two other features of this tool are worthy of note. First, it also performs the related optimization of transforming division by a constant (or more generally, any expression with a known fixed value) into a multiplication by the (pre-computed) reciprocal of that constant, since multiplications are considerably faster to compute than divisions. Second, if the input model is dimensionally consistent, then the output model will also be dimensionally consistent. New units elements are added to the generated model where necessary.

Another domain-specific optimization is the use of lookup tables to precompute the values of expressions that would otherwise be repeatedly calculated. Several expressions in most cardiac electrophysiological cell models contain only one variable: the membrane potential, \( V \). They also typically contain exponential functions that are expensive to compute. Under normal physiological conditions, the membrane potential \( V \) usually lies between \(-100\) mV and \(50\) mV, and so a table can be generated of pre-computed values of each suitable expression for potentials within this range. Then, given any membrane potential within the range, a value for each expression can quickly be computed using linear interpolation between two entries of the lookup table, which is faster than computing an exponential directly. The technique has been in use for some time (Dexter et al. 1989), with the lookup table code hand written for each equation. The analysis can, however, be automated for models described in CellML (Cooper et al. 2006), and a prototype tool is included in PyCml.
Note that the use of linear interpolation means that this optimization does introduce an accuracy penalty. Currently, no automatic analysis is done to estimate whether the magnitude of this is acceptable, but it has not been an issue in practice. Work on producing a computable error bound is ongoing.

Through automating optimizations such as these that have traditionally been applied by hand, it is our opinion that the simulation performance of models described in CellML can be comparable to hand-optimized versions. Research into coupling CellML models within tissue-level simulations in an efficient manner is also progressing well. There will thus soon be little reason not to use mark-up languages for model representation.

(e) Simulation

The CellML tools page (see http://www.cellml.org/tools/) currently lists the following six CellML simulation environments: AGOS; Cell Electrophysiology Simulation Environment 1.4.7 (CESE); COR 0.9.31.901; JSim 1.6.79; PCEnv 0.3.1; and VCell 4.3.1.

All environments were tested on a ThinkPad T61p laptop computer (Intel Core 2 Duo CPU T7500 @ 2.20 GHz) running a ‘clean’ 32-bit Windows XP Professional Service Pack 2 system under VMWare 6.0.2.59824 using 1 GB of dedicated RAM. AGOS was tested using the AGOS Web server, an AthlonX2 @ 2.0 GHz with 1 GB of RAM running Kubuntu Linux.

These environments were tested against the CellML 1.0 of classic and widely used cardiac electrophysiological models (table 2), with the aim of comparing the relative speed of each environment.

The models were successfully read by both COR and PCEnv, and successfully imported by AGOS. JSim was able to import the Noble (1962) and Garny et al. (2003b) models, but not the other models: a MathML issue prevented JSim from successfully importing them. Neither CESE nor VCell was able to import any of the models. We understand, however, that further work is needed in the import feature of these last two applications.

AGOS, COR, JSim and PCEnv provide access to different integrators, including CVODE that is part of the SUNDIALS library (see http://www.llnl.gov/casc/sundials/; Hindmarsh et al. 2005) and comes with a choice of methods, iterators and linear solvers which are ideally suited for biological problems. AGOS, PCEnv and JSim offer two sets of settings suitable for both stiff and non-stiff problems, respectively, and allow the user to change the relative and absolute errors (and the maximum time step for PCEnv), while COR allows full customization of CVODE. VCell offers a wide range of integrators, including LSODA a precursor of CVODE, and it is not clear which integration method CESE uses.

CVODE was set to solve stiff problems (i.e. a BDF method, a Newton iterator and a dense linear solver, in the case of COR). In all cases, the relative and absolute errors were set to $10^{-7}$ and $10^{-9}$, respectively. The duration of the simulation (table 2, column 2) was set to get a computational time of the order of seconds when sampling the results every millisecond (i.e. 1 kHz) using the fastest environment. Some of the models require an electrical stimulus that is characterized by a duration that must be fed into CVODE by setting its maximum (usable) time step (table 2, column 3). Where no stimulus was required,
The maximum time step was set to 0 ms in COR (i.e. no maximum time step) or to the duration of the simulation in PCEnv (allowing PCEnv to compute the model as fast as possible).

Seven simulations were run without generating any graphical output, of which the two slowest were discarded (to account for unpredictable slowdowns of the machine). Timings for AGOS, COR and JSim were provided by the modelling environment itself, while we had to manually time PCEnv, making the results

Table 2. Simulation results. (Cardiac electrophysiological models of different cell types (see footnotes) and from different authors (column 1) were tested. The simulation duration of each model (column 2) was chosen based on the model’s computing requirements while the value of its stimulation duration (where applicable) was provided by its CellML file. The last four columns list the normalized (with respect to COR) time taken by AGOS, COR, JSim and PCEnv, respectively, to compute each model without and with graphical output.)

<table>
<thead>
<tr>
<th>cell model</th>
<th>simulation duration (s)</th>
<th>simulation duration (ms)</th>
<th>AGOS</th>
<th>COR</th>
<th>JSim</th>
<th>PCEnv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Garny et al. (2003b)^</td>
<td>100</td>
<td>n.a.</td>
<td>3.45/4.86</td>
<td>1/1</td>
<td>27.15/111.59^</td>
<td>3.85/4.18</td>
</tr>
<tr>
<td>Courtemanche et al. (1998)^</td>
<td>100</td>
<td>2</td>
<td>3.38/4.72</td>
<td>1/1</td>
<td>3.92/4.18</td>
<td></td>
</tr>
<tr>
<td>Nygren et al. (1998)^</td>
<td>100</td>
<td>6</td>
<td>6.01/8.37</td>
<td>1/1</td>
<td>4.79/4.90</td>
<td></td>
</tr>
<tr>
<td>Noble (1962)^</td>
<td>1000</td>
<td>n.a.</td>
<td>4.28/3.78</td>
<td>1/1</td>
<td>26.48/166.09^</td>
<td>2.67/2.35</td>
</tr>
<tr>
<td>Luo &amp; Rudy (1991)^</td>
<td>200</td>
<td>2</td>
<td>3.96/4.69</td>
<td>1/1</td>
<td>4.06/3.90</td>
<td></td>
</tr>
<tr>
<td>Noble et al. (1998)^</td>
<td>100</td>
<td>3</td>
<td>4.54/6.42</td>
<td>1/1</td>
<td>4.71/5.14</td>
<td></td>
</tr>
<tr>
<td>Faber &amp; Rudy (2000)^</td>
<td>50</td>
<td>2</td>
<td>3.74/5.58</td>
<td>1/1</td>
<td>4.59/4.92</td>
<td></td>
</tr>
<tr>
<td>Bondarenko et al. (2004)^</td>
<td>10</td>
<td>0.5</td>
<td>4.11/4.99</td>
<td>1/1</td>
<td>4.43/4.58</td>
<td></td>
</tr>
<tr>
<td>ten Tusscher &amp; Panfilov (2006)^</td>
<td>100</td>
<td>1</td>
<td>3.34/4.51</td>
<td>1/1</td>
<td>5.35/5.83</td>
<td></td>
</tr>
<tr>
<td>average</td>
<td>4.09/5.32</td>
<td></td>
<td>26.82/138.84</td>
<td></td>
<td>4.26/4.46</td>
<td></td>
</tr>
</tbody>
</table>


^bOwing to the slow rendering of the simulation data, the simulation was run only once.

^cHuman atrial cell. See [http://www.cellml.org/models/courtemanche_ramirez_nattel_1998_version03](http://www.cellml.org/models/courtemanche_ramirez_nattel_1998_version03). The stimulus protocol of the model was altered, so that it could generate a periodic stimulus for the duration of our test simulation.


^fOwing to the slow rendering of the simulation data, the simulation was run only once and for 100 s worth of cardiac electrical activity, and the simulation time extrapolated.


^kHuman ventricular. The M cell version of that model was used (as shipped with COR). A ‘proper’ stimulus protocol was added to the model, so that it could generate a periodic stimulus for the duration of our test simulation.
slightly less accurate (by a few tens of a second at the very most). The same protocol was repeated while plotting the membrane potential (one of the key parameters of these models) against time. To make the comparison meaningful, the plotting area was made to be of the same dimensions in all environments.

Results for AGOS, COR, JSim and PCEnv are shown in the last four columns of table 2. The first number corresponds to the (normalized with respect to COR) time taken to complete the simulation without generating any graphical output, while the second number is for the case where the membrane potential is plotted against time.

In all cases, COR proved to be the fastest both with and without graphical output. In the case where no graphical output was generated, AGOS was second fastest (approx. 4.09 times slower), while PCEnv third (approx. 4.26 times slower) and JSim last (approx. 26.82 times slower). With graphical output, PCEnv was second fastest (approx. 4.46 times slower), while AGOS third (approx. 5.32 times slower) and JSim last (approx. 138.84 times slower).

All four environments use the same integration solution, highlighting major differences in the way they handle simulations. On the computational side, JSim and PCEnv (and VCell) keep track of all computed data, so that the user can access any of them once the simulation is completed. AGOS and COR (and CESE), however, request the user to select the data to track. On the graphical side (see figure 3 for a typical output from the different simulation environments tested in this review), AGOS (and CESE and VCell) render the simulation data once the simulation is finished while COR, PCEnv and JSim render them in ‘real time’. COR renders the results at twice the vertical frequency of a computer screen (120 Hz in the present case), giving the illusion that the results are plotted as they are being computed. JSim and PCEnv, on the other hand, plot the results in chunks (i.e. whenever enough data have been generated). Also, PCEnv does not render the simulation data at regular intervals, but a maximum number of them per simulation (we set that parameter to a value that would make it equivalent to rendering the data every millisecond). There are obviously advantages and disadvantages to those different approaches, but these are beyond the scope of this review.

Discrepancies were also found in the results generated by the different environments, and this despite using the same CellML models and CVODE parameters. Figure 4 illustrates some of those discrepancies for the Noble (1962) and ten Tusscher & Panfilov (2006) models. Results for the Noble (1962) model are virtually identical for AGOS, COR and PCEnv (figure 4b), but those for JSim are shifted by approximately 9 ms (figure 4a). In the case of the ten Tusscher & Panfilov (2006) model, both AGOS and COR still yield virtually identical results while PCEnv’s results are different during the repolarization phase (an important phase when, for example, studying cardiac arrhythmia), as illustrated in figure 4c. It would be difficult to explain this last result without further investigation, but once again this is beyond the scope of this review.

(f) CellML API

An API is the interface that an application provides in order to allow requests for services to be made of it by other computer programs. It describes how software developers may access a set of functions without requiring access to the source code of the functions or library, or requiring a detailed understanding of

Phil. Trans. R. Soc. A (2008)
the internal workings of the functions. The software that provides the functionality described by the API is said to be an implementation of the API. The API itself is abstract, as it is an interface.

The CellML API provides a simple interface that applications can use to manipulate and process CellML documents. The interface is designed to be independent of any programming language, platform or vendor, and is expressed in Interface Definition Language. It addresses both CellML 1.0 and 1.1, and
Figure 4. Simulation comparisons of the Noble (1962) and ten Tusscher & Panfilov (2006) models. The last second of the Noble (1962) and ten Tusscher & Panfilov (2006) simulations are shown in (a,b) and (c,d), respectively. (a,c) The membrane potential of those models using the various environments. (b,d) The difference in membrane potential between the various environments and COR (COR was selected as a reference on the sole basis that it is the fastest environment; linear interpolation was used to convert the raw PCEnv data to 1 kHz data).
supports metadata. The API is an object model-based system such that the entire model is parsed to create the top-level object. Once the top-level object has been created, the Core API describes methods to parse, select, create and transform CellML documents in an object-oriented manner. It allows event consumers to register and receive notifications when certain parts of the underlying CellML model are modified.

Other modules can be turned on at configuration time to provide further functionality. For example, the Context module keeps track of a hierarchy of models, as well as generalized annotation data. It also tracks which tools and services are open. This allows multiple tools to interoperate on a single system with a single list of open models. Other modules include CCGS, which generates C code for a CellML model, and CIS, which integrates CellML models (representing systems of ODEs).

4. Future directions

The future of computational physiology (one could argue physiology itself and, more generally, integrative biology) depends on our ability to create a computational science infrastructure that facilitates biophysically based model development, multi-scale modelling, model and data sharing and an ability to customize models to a particular species and to an individual within that species. CellML and its associated mark-up languages are an important contribution to that computational framework. Here, we propose future directions for the mark-up languages, model repositories and metadata and discuss the application of the IUPS Physiome Project to health care.

(a) Mark-up languages

(i) Data typing in CellML

CellML 1.0 and 1.1 use a very simplistic typing scheme (all variables are currently assumed to have only real scalar values). While this approach has a certain elegance in its simplicity, it does sometimes force the model author to adopt verbose or awkward representations of a variable. For example, variables that are intrinsically discrete in nature (e.g. numbers or items) are forced to be represented as a continuous quantity. Furthermore, variables that are structured (e.g. tensors) must be represented in component form as separate real variables.

CellML uses MathML as the description language for representing mathematical associations between variables. MathML already has a rich set of scalar (integer, real, rational, complex Cartesian and complex polar) and structured (set, list, vector and matrix) types. It seems sensible to enable CellML to take advantage of the typing mechanism available in MathML. In theory, this would not compromise the mathematical expressions expressed in CellML, since MathML is already aware of typed variables. The major issue, here, is that CellML currently has no mechanisms for specifying or checking the type of variables. In order to include such mechanisms, the type of any variable would have to be specified within the variable element, so that sufficient typing information is available at the component interface to enable type checking when variables in different components are associated using the connection element. Consideration is

Phil. Trans. R. Soc. A (2008)
currently being given to how such typing mechanisms might be added to CellML. Clearly, any such addition must not compromise existing features (e.g. units checking) or the underlying MathML representation of mathematical expressions.

CellML represents the relationship between variables declaratively, as opposed to imperatively. In this sense, the relationships have no sense of sequence (each equation must be satisfied simultaneously with every other). The absence of sequence makes it difficult to express the relationships that effect a change of topology or state. Fortunately, the addition of structured types offers a solution to this problem. By representing variables as lists or vectors, where the index represents successive states, one can express variables at a future state in terms of variables at past states. For example, a variable \( v \) at state \( n \), represented by the vector component \( v(n) \), may be expressed in terms of variables at earlier states, \( v(n-1) \), etc. Structured types thus provide a simple way of expressing sequences without compromising the declarative nature of the underlying mathematical representation.

(ii) FieldML for spatial field modelling

Despite the name, CellML is not confined to cellular and subcellular models. It is just as applicable, for example, to systems physiology models that deal with lumped parameter descriptions of a whole organ or organ system. The XML standard being developed for encoding spatially varying fields is FieldML (see http://www.physiomeproject.org/xml_languages/fieldml/). This is, for example, used to hold the parametric description (usually in the form of finite-element meshes) of the geometry and structure of an organ (including, for example, the distribution of protein density), the spatial distribution of material properties and the boundary and initial conditions required for the solution of a physical problem in computational physiology. Following the solution of the partial differential equations governing the physical problem, the resulting solution fields are output as FieldML (e.g. the time-varying membrane potential distribution and mechanical deformation in a beating heart, the oxygen concentration in a muscle, the stress and strain distributions in a bone). Note that FieldML can be used for defining the spatial variation of parameters within a CellML model or a CellML model can be called at a large number of material locations within a tissue model with spatially varying properties.

(iii) ModelML for boundary-value problems

At present, the physical equations governing a boundary-value problem in computational physiology are encoded in the computational code (usually written in C, C++ or Fortran). If, in the future, disparate groups working on different aspects of an organ system wish to merge their model equations into a composite model in order to examine more complex integrative system behaviour, it will be necessary to develop a mark-up language (tentatively called ‘ModelML’) for the equations themselves. The strategy being adopted here is to develop a library of computational code that implements the standard spatial field operators such as \( \nabla \) (gradient), \( \nabla \cdot \) (divergence), \( \nabla \times \) (curl) and \( \int d\Omega \) (domain integration). Many of the terms in the equations governing the physics of tissue and organ function (reaction–diffusion, fluid mechanics, finite deformation elasticity, etc.) can be expressed in terms of these operators. The mark-up language...
will then specify how these components are incorporated into the composite
governing equations. Since in many cases the models being combined will deal
with different spatial scales, it will be necessary for ModelML to specify the multi-
scale linkages.

(b) Model repository

(i) Component libraries

The import element of CellML 1.1 provides a very powerful mechanism to
build libraries of generic or reusable models and components. It is clear that this
feature can enable rapid and robust model building by providing libraries of
frequently used models, such as generic descriptions of reaction kinetics. The
creation of such libraries has, however, so far not yet received much attention,
though some examples can be found at http://www.cellml.org/models/
international_si_units_2006_version01 and http://www.cellml.org/models/
mohr_taylor_newell_2008_version03.

(ii) Model curation standards

An important step for the CellML project is the development of curation
standards and to make the biophysical limitations of the models apparent. The
authoring, editing, validating and simulation codes, such as the ones discussed in
this review, and the Web interfaces that allow uploading of curated CellML
models have now reached the point where a focused effort on curation of the
models in the CellML model repository can be undertaken. To achieve level 3
curation will require the development of further tools (e.g. to analyse charge
conservation in cardiac electrophysiology ion channel models and mass
conservation in protein pathway models).

(c) Metadata

While CellML provides a way to specify a mathematical model such that it can
be shared between different research groups and toolsets, further information is
often required to be able to use the model correctly and consistently.

It would be extremely useful if model authors could provide information in a
machine readable format, not just about their model, but also about the
particular simulation they have run. In this way, it becomes possible to
reproduce any simulation results obtained by the authors simply by loading
the information into a simulation tool and asking it to rerun the simulation.
To this end, the CellML team is developing some simulation metadata
specifications defining a standard, machine readable method for specifying
the information required to perform a specific simulation using a given CellML
model (see http://www.cellml.org/specifications/metadata/simulations/). The
simulation metadata defines such things as the range over which independent
(bound) variables should be integrated, the specific numerical methods to use
and any special parameters that such methods require.

In order to accurately reproduce given experiments, it is important to know
not only how to perform the required simulations but also how to interpret the
simulation output data. To begin addressing this issue, the CellML team is
developing some graphing metadata specifications that define how simulation
outputs can be drawn as graphs (see http://www.cellml.org/specifications/
metadata/graphs/). The initial focus on graphs follows their extensive use in the
current literature as the primary method for demonstrating model validity. The
graphing metadata will define not only how to draw a given graph using results
from a possibly wide range of simulations, but also include links to experimental
data sources. While this provides a way to automatically draw graphs containing
both numerical simulation results and experimental data suitable for publication,
the same metadata can be used to other ends (e.g. numerically compare
simulation results to experimental data).

Such metadata will provide the means to perform (and automate) the type of
validation touched on at the start of §3b, and aid in level 3 curation.

(d) Application to health care

The development of CellML, FieldML, ModelML and their associated tools
and model repositories is an important part of the IUPS Physiome Project. They
are providing a robust framework for handling mathematical models of
physiological processes in an unambiguous fashion and capturing biophysical
mechanisms at multiple spatial scales with quantitative models. However, the
IUPS Physiome Project and the complementary European initiative called
the EuroPhysiome or Virtual Physiological Human (see http://www.europhy-
siome.org/) project, described elsewhere in this issue, have a much grander
vision: that of linking the bioinformatics world of genomics and proteomics to the
clinical world of patient-specific diagnostic medicine, surgical planning, virtual
surgery, drug discovery and implant design. Here, we discuss some of the future
developments that are needed to bring the current Physiome modelling
framework to bear on clinical medicine.

Figure 5 shows a schematic of how a clinical data workflow might make use of
the modelling framework.
In this scenario, clinical measurements at the physiological level are fitted with CellML systems physiology models using parameter estimation tools to give patient-specific models. Clinical measurements of organ function from medical imaging devices such as magnetic resonance imaging (MRI) and computerized tomography (CT) are used together with population atlases to give model templates that are also fitted to yield patient-specific organ models. Clinical genomic and proteomic data are used with CellML pathway models and other subcellular models to produce patient-specific cell models. The cell models are used in organ computations to inform the parameters of the systems physiology models and hence to inform the clinical decision.

Several further developments are also required to apply these mark-up language-based model repositories to the interpretation of clinical data. In particular, there is an urgent need to develop standards for encoding clinical data and associated metadata and to link the models, via the metadata, into clinical ontologies such as SNOMED (see http://www.snomed.org/) now being developed by the International Health Terminology Standards Development Organization (see http://www.ihtsdo.org/). The models will also need to deal with statistical variation among populations and with parameter changes associated with pathologies.

5. Conclusion

Since its release in 2001, several hundred models have been encoded in CellML. Not only does this provide model users with a comprehensive repository of models, but also models that address a wide range of problems (e.g. electrophysiological, mechanical, biochemical), illustrating the versatility of the CellML language. However, without the right tools and techniques, CellML models would be of limited use.

The editors, automatic code generators, GUI environments, Web portal, simulators and repository presented in this review provide powerful solutions for computational physiology, but this is only one step in the right direction. The CellML specifications need refining and so do the CellML resources. In addition, there is a need for the release of complementary mark-up languages, such as FieldML and ModelML.

These mark-up languages, associated tools and techniques, as well as further developments are essential to the strategy of the IUPS Physiome Project for building a descriptive, integrative and predictive framework for the modelling of the human body.

A.G. is supported by a grant from the UK Biotechnology and Biological Sciences Research Council (BB/E024955/1). R.W.d.S. acknowledges the support provided by the Brazilian CNPq, CAPES and FAPEMIG.

References


