THE ROYAL SOCIETY

*doi* 10.1098/rstb.2001.0910

# Towards NeuroML: Model Description Methods for Collaborative Modelling in Neuroscience

**Nigel H. Goddard[1*], Michael Hucka[2], Fred Howell[1], Hugo Cornelis[3], Kavita Shankar[2] and David Beeman[4]**

[1]*Institute for Adaptive and Neural Computation, Division of Informatics, University of Edinburgh, 5 Forrest Hill, Edinburgh EH1 2QL, Scotland*

[2]*Division of Biology 216-76, California Institute of Technology, Pasadena, CA 91125, USA*

[3]*Theoretical Neurobiology, Born–Bunge Foundation, University of Antwerp, Universteitsplein 1, 2610 Wilrijk, Belgium*

[4]*Department of Electrical and Computer Engineering 425 UCB, University of Colorado, 425 UCB, Boulder, CO 80309, USA*

Biological nervous systems and the mechanisms underlying their operation exhibit astonishing complexity. Computational models of these systems have been correspondingly complex. As these models become ever more sophisticated, they become increasingly difficult to define, comprehend, manage and communicate. Consequently, for scientific understanding of biological nervous systems to progress, it is crucial for modellers to have software tools that support discussion, development and exchange of computational models. We describe methodologies that focus on these tasks, improving the ability of neuroscientists to engage in the modelling process. We report our findings on the requirements for these tools and discuss the use of declarative forms of model description—equivalent to object-oriented classes and database schema—which we call templates. We introduce NeuroML, a mark-up language for the neurosciences which is defined syntactically using templates, and its specific component intended as a common format for communication between modelling-related tools. Finally, we propose a template hierarchy for this modelling component of NeuroML, sufficient for describing models ranging in structural levels from neuron cell membranes to neural networks. These templates support both a framework for user-level interaction with models, and a high-performance framework for efficient simulation of the models.

**Keywords:** NeuroML; computational models; simulation; databases; interoperability

## 1. INTRODUCTION

The complexity of neural systems induces corresponding complexity in the models created to describe and explain them. We are interested in developing methodologies which will improve the ability of neuroscientists to collaborate in the modelling process. There are two key technical requirements for supporting this. First, we must have technologies that enable the exchange of simulation software components, and we have been developing such methods over the last few years (Goddard *et al*. 2001). Second, it is crucial for modellers to have access to software tools that support discussion, development and exchange of computational models and their components, including databases of models (Beeman *et al*. 1997). In this paper, we report our findings on the requirements for these tools and our proposals for organizing their development. The key result we report is a mark-up language for describing models of neural systems and processes in a form suitable for use by simulation software tools. This mark-up language is a component of a larger collaborative effort to produce NeuroML, the Neural Open Mark-up Language, an open standard for describing models, data, literature, methods and other aspects of neuroscience.

*Author for correspondence (nigel.goddard@ed.ac.uk).

### (a) *Barriers to discussing and sharing models*

What is the fundamental problem confronting designers of methods for describing computational models in neuroscience? A computational model of a physical system, whether it is a synapse, a neuron or a network of neurons, is in principle a concretization of a conceptual structure held by a scientist or group of scientists. To enable simulation by a computational engine, the model must first be elaborated and expressed in a formalized way, usually in a computer program. This process of elaboration and formalization often produces a description which obscures a model's essential components and structure.

Researchers would be better served by means of expression that expose the important structural components of a model, as well as support discussion and exchange of models. But to date, several factors have been barriers to this goal. (i) The often *ad hoc* nature of the mapping from conceptualization to formalization can render the textual description opaque: it can be difficult to understand which aspects of the model's concrete manifestation (e.g. which elements in a given simulation script) express the important conceptual structures. (ii) Many different programming and scripting languages can be used to express a given model. Different simulation engines with different capabilities are often restricted to using their own single expressive language whose form is substantially different from those used by

 

other simulation engines. A consequence of this is that modellers familiar with one simulation environment find the formalizations of another environment too obscure. (iii) Different formalizations do not usually mix well, and support for multiple formalizations is rarely present in existing simulation environments.

These are some of the factors that have led to a situation which mitigates against discussion, shared development and exchange of models.

### (b) *Overcoming the barriers*

To begin addressing the problems listed above, we propose a set of conceptual schemas for constructing model descriptions. A novel aspect of our approach is to push model descriptions in the direction of a declarative form, enabling models to be described by database schemas and simulated by encapsulated code modules. This has a number of beneficial implications. First, it permits models composed of well-understood and accepted structures to be described simply by *naming* these structures. Second, it enables the creation of database schemas that correspond to the named structures. Third, it enables simulation code modules that generate and execute these structures to be invoked using the names. And finally, it permits purely declarative descriptions of such models (e.g. 'there exists an X of type Y with parameters P1, P2, ... P*n*').

While adoption of a declarative form of model description is an important step, it is not sufficient. Research models often incorporate novel structures which cannot be described by an existing schema. Novel structures must be described in complete detail. Often the most effective way to do this is to use a programming language which has the imperative programming concepts of sequentiality (A then B then C), iteration (while ... do ...), and conditional branches (if ... then ...). However, imperative descriptions are undesirable from the standpoint of model exchange: they require much greater effort to understand than declarative forms, and database search engines can make no effective inferences about imperative program behaviour.

We avoid this conflict by ensuring that imperative forms of description (i.e. programming languages) are available in the representation framework, but that these imperative forms are encapsulated in code modules which have well-defined interfaces. An interface takes the form of a declarative description, and can therefore be used as part of a database schema.

### (c) *Overview*

In § 2, we examine some of the requirements for model descriptions which have motivated this work. In § 3, we review previous forms of model descriptions, and in § 4, we elaborate on the desirability for the separation of declarative and imperative forms. We discuss in § 5 the relationships between declarative model descriptions, database schemas and object-oriented classes, using the term *template* to refer to these concepts, and we introduce the model component of NeuroML as the declarative form of model descriptions which can be generated from templates; this can serve as a medium of information exchange between software components. In § 6, we describe in detail the templates which structure generic, neuronal network, single-neuron and cellular mechanisms, respectively. Finally, in § 7 we summarize two software frameworks that are currently being implemented using NeuroML: NEOSIM for simulation, and Modeler's Workspace for databases and user interfaces.

## 2. MODEL DESCRIPTION REQUIREMENTS

Computational modelling in neuroscience is carried out for a variety of reasons. In our view, the most fundamental purpose is elaborating and testing hypotheses derived from conceptualizations of neural system functions. The development of conceptual structures lies at the heart of the scientific process. This development usually proceeds using a combination of introspection, discussion with other scientists, and experimentation. Discussion traditionally involves an iterative cycle of informal meetings, workshops, conferences and, finally, the published literature. Experimentation usually involves testing predictions (hypotheses) generated from a conceptual structure (theory) by gathering empirical evidence.

Computer simulations may be useful in cases where the consequences of a particular conceptualization are not immediately apparent or the goal is to test rigorously whether the conceptualization produces a particular prediction. Creating simulations requires scientists to express their conceptualizations in a form sufficiently rigorous and concrete that the results can be executed as computer programs. (This is itself a highly illuminating exercise which can shed light on deficiencies in the conceptualizations.)

We refer to these concrete forms as *model descriptions*, and have identified three key characteristics that we believe are required for a model description language to be viable.

(i) *Clarity.* The process of concretization is often difficult, with the consequence that the technical difficulties of expressing the ideas concretely (e.g. in a programming language) can obscure the structure being expressed. To be effective, the concrete form must be easy to work with, so that these technical difficulties do not dominate the process and so that the conceptualization being expressed is clear to humans.

(ii) *Portability.* Simulation by computer is only possible if the concrete form is expressed in a programming language or can be translated into programming language form. Concretizations are sometimes expressed at the level of a traditional programming language (e.g. Fortran or C), but often, in an attempt to render the concrete form more transparent and easier to work with, it is not a full programming language but something which can still be translated into a program. These have often been simple script languages or descriptive forms. For a variety of reasons, some discussed below, a large number of such concrete representation formats have come into existence (one per simulation environment) in the domain of computational neuroscience. This makes it difficult to transfer or share models outside the community that uses a particular simulator. Portability is therefore an important requirement for the usefulness of a model description language.

(iii) *Modularity.* As theoretical work in neuroscience grows, it is clear that simulations will continue to play an increasingly significant role. The systems we deal with in neuroscience are complex, structured, heterogeneous and often poorly understood, leading to the same qualities in the conceptualizations we develop to explain these systems. A huge range of spatial and temporal scales are involved: perhaps six orders of magnitude in the spatial dimension (microns to metres) and fifteen in the temporal dimension (microseconds to decades). Researchers focus on different phenomena observable at a variety of spatial and temporal scales. The need to place their models in a broader context and examine interactions with phenomena at other spatial and temporal scales pushes researchers to collaborate and share modelling efforts. Model description methods can only support this kind of collaborative, distributed work if they are truly modular.

In the course of over 15 years of developing simulation environments for various levels of modelling in the neurosciences (Goddard 1994; Goddard & Hood 1997; Hines & Carnevale 1997; Bower & Beeman 1998), we have come to regard these three requirements as key for a model description framework to be able to support the evolving needs of computational neuroscientists. Recent interest in creating databases of models (Peterson *et al*. 1996; Mirsky *et al*. 1998; Shepherd *et al*. 1998) and in using models to organize information in databases (Beeman *et al*. 1997) has led us to refine these ideas. The following section describes how some previous and current modelling environments address these requirements, and the subsequent sections describe our proposals for the next generation of tools.

## 3. MODEL DESCRIPTIONS PAST

Computational modelling environments for neuroscience typically evolve from the needs of an individual research project; it is rare that these environments are designed from the beginning with the intention of addressing a wide variety of modelling needs. Rather, the environment is greatly conditioned by the research project it supports. When the time comes to extend the capabilities of the simulation environment, the original design is often found to preclude certain desirable options.

Most environments eschew general-purpose programming languages in favour of domain-specific script languages for model description. There are three reasons for this. First, most programming languages that are considered for the task must be compiled, whereas script languages are interpreted. Compilation usually increases the time required to make and evaluate a small change in the model, so the compilation step itself introduces some delay. For all but the smallest models, the more significant factor is that compilation without dynamic linking requires that the model be reconstructed in the computer. Because model construction can take as much time as running a simulation, this can be very costly. For example, in a large-scale simulation of the cerebellar network (Howell *et al*. 2000), we found that use of a parallel computer was essential merely to accommodate the size of the model. At the degree of parallelism used in that study (128-way), with execution of the model running much faster than on a single processor, we found that model construction took as long as the model runtime, even though model construction itself was being performed in parallel.

The second reason to avoid general-purpose programming languages is that they have both too much textual baggage in their descriptions (at least to the non-expert) and too little expressive support for concepts considered basic by computational neuroscientists (e.g. 'ion channel'). This applies to general-purpose script languages as much as to compiled languages. The preferred solution is to design a script language that is closely tied to the domain under study and that has enough expressive power to support the range of model descriptions considered likely (e.g. the HOC language used in NEURON (Hines & Carnevale 1997) or the SLI language used in GENESIS (Bower & Beeman 1998)). The third, related reason to avoid general-purpose languages is that it is often unclear to the developers of simulation environments which languages will continue to gain support and be maintained in the future.

For these and other reasons, there now exist a number of special-purpose script languages. Unfortunately, because most simulation environments originated in relatively small, focused projects, attempts to extend the special-purpose languages to handle larger and more diverse projects has not been easy. Most of them provide neither the range of neuroscience-specific constructs needed, nor the expressive power (e.g. object-orientation) that larger and more diverse projects require. This has typically led to modifying the script languages to add extensions, rather than replacing the languages outright, in order to support existing and legacy models. But this is often impossible to accomplish cleanly, with the result that the script language turns into a not-quite-general programming language with some, but not all, of the required neuroscience-specific constructs.

## 4. MODEL DESCRIPTIONS FUTURE

Our understanding of the role of a model description language in the modelling process and the design of computational tools has evolved over many years of experience in modelling and software design. We now think of the description language as primarily a medium for exchange of information between software components in a computational system, but with the crucial caveat that it must directly support the software components that allow the scientist to describe, visualize and interrogate the model under study.

The kind of software architecture we envisage to support computational modelling is shown in figure 1. An example of a system employing this architecture is the Systems Biology Workbench (Hucka *et al*. 2001*a*). In this approach, a core plug-in manager loads software components on demand. Components may include interfaces to databases, simulation environments, model-specification user interfaces, visualization interfaces, and others. While some components may exchange specialized information in private forms, and some may implement their own plug-in capability for specialized purposes (e.g. the NEOSIM

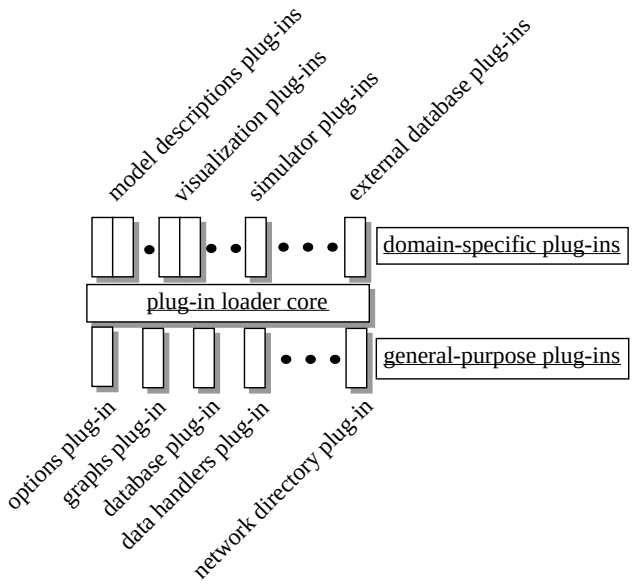


Figure 1. The component-based architecture: many components use NeuroML to communicate. The same plug-in loader is used for domain-specific plug-ins as for the library of general purpose plug-ins.

framework for simulation plug-ins), the main medium of exchange of information between components is the description language (e.g. NeuroML). The main difference, then, between our current view of model description languages and previous model description languages is that we do not anticipate direct manipulation by modellers of the textual form of the language. Rather, we expect that user-interface components will provide graphical and other means for model description, and that this will be translated into the model-description language. Thus, the model-description language need not be easily readable by modellers, but it *must* support the components with which modellers interact, and must support extensions that enable additional components describing new conceptualizations. What, then, are the specific requirements for this kind of model-description language?

### (a) *Requirements*

Let us go back and re-examine the arguments for and against domain-specific script languages versus general-purpose programming languages for model descriptions, and evaluate them in light of the three requirements laid out earlier: clarity, portability, and modularity.

(i) *Requirement 1. Clarity*

Clarity in a description language involves a balance between three objectives. First, the description language should support, as closely as possible, the language that neuroscientists use in verbal, textual and graphical descriptions during scientific discourse. Second, the language should be designed in such a way that it can be translated easily into a computer program which can run efficiently on a variety of computing platforms. Third, it is essential that the description language be precise, meaning that there can be only one interpretation of any given statement in a particular context.

Fortunately, these objectives do not conflict. The first objective suggests that the language should incorporate well-established conceptual structures under specific names (e.g. 'Hodgkin–Huxley channel', 'topographic projection'), and should support the ability to add to the set of named structures (e.g. 'My-Hypothesised channel', 'LGN-V1 projection'). The second objective can also be served by incorporating high-level conceptual structures into the model description language. For example, a description of a population of neurons given as

```
DefinePopulation(Population-name, CellType, Number)
```

can be translated into a variety of forms suitable for different computational platforms, including parallel platforms, whereas a description of the form

```
for i = 1 to Number
  DefineCell(Celltype)
```

is not amenable to a similarly wide range of optimizations and, in particular, is not as suitable for parallel platforms. The first form can be executed in parallel if the DefinePopulation primitive is suitably implemented. The second form is inherently sequential in that, without analysing the loop, it is impossible to know whether the neuron descriptions are independent.

The objective of well-definedness can be met by ensuring that the constructs added to the language have precise semantics. In particular, this means having clear definitions of how high-level, domain-specific constructs interact with each other. From the modeller's perspective, much of the substance of making the model description well-defined inheres in the understanding of these high-level constructs; e.g. what exactly is meant by a 'topographic projection'? This understanding requires that such constructs be documented adequately in appropriate forms (e.g. textual, pictorial, movies).

(ii) *Requirement 2. Portability*

The portability of a model description is a matter both of transfer from one computational platform to another and of transfer from one simulation environment to another. Most of the aspects of transfer from one computational platform to another can be reduced to the question of whether an appropriate simulation environment (i.e. one capable of executing the model description) exists on the target platform. One consideration which is not merely an implementation issue at this stage of technical development is the extent to which a model description is amenable to parallelization. A model description is parallelizable if it is expressed in a way that allows efficient algorithms both to partition the model across multiple processors and to construct the model in the computer in parallel. One way in which descriptions can be made more likely to be parallelizable is by expressing them at the highest conceptual level; then, the implementation can vary depending on the type of computational platform. Although this is not sufficient in and of itself, it is a good heuristic.

The ability to transfer a model description from one simulation environment to another is the major portability issue. Given that for the foreseeable future there will continue to be a diversity of simulation environments, the utility of a database of model descriptions will depend on their portability between environments. Generally speaking, portability between two environments is only

possible if the languages they use can be unambiguously translated in both directions without loss of information, and if this is the case for a number of environments then there must exist a common language they can all use (trivially, the language used by one of the environments!). Consequently, the most practical approach to enabling transfer between environments is to seek a common language.

The essential requirement for portability, then, is the availability of a common model-description language which has extensibility built into it from the start. In fact, we should think of it as an evolving language. This language can be the basis for exchange of models between research groups, and for exchange of information with databases of models.

(iii) *Requirement 3. Modularity*

As models become ever more complex and span more structural levels, it is important that model descriptions become modular: as model descriptions become modular, we find that the associated language itself becomes modular—that is, we require language elements focused on particular problem domains, sharing a common basis. For example, we may wish to describe channel kinetics using equations, so the language element dealing with channel modelling should support the expression of equations. But we may also wish to describe channels using some well-known form (e.g. dual-exponential) in which we use a name that stands in for particular equations. The network-level language element needs to be able to express the concept of populations of cells and projections between populations, perhaps in a hierarchical manner. We may also wish to describe networks in terms of equations governing the interaction between cells, perhaps on a spatial basis (e.g. long- versus short-range influences). If the underlying language common to all language elements supports expression of equations, we can specialize this in each language element to support the particular needs of models of processes at a given structural level.

Language elements also provide advantages for the other two requirements of clarity and portability. Smaller languages tend to be more easily comprehensible, and restricting the range of legal expression to that appropriate for modelling particular types of processes ensures that extraneous constructs and terms do not obscure the model description. Portability is more easily assured because it is only the tools associated with a particular level of modelling that need to be available in the target simulation environment. This allows for incremental enhancement of a simulation environment: if a capability needed for a particular language element is missing, it can be added to make the language element available in that environment. If, however, we used a single monolithic language, then every aspect of that language would need to be available before any particular (and much smaller) subset were usable for a particular modelling project.

### (b) *Program-free model descriptions*

We have established that we want a common language framework within which we can design extensible language elements focused on particular subdomains of the neuroscience modelling endeavour. We want these languages to incorporate abstractions corresponding to the conceptual structures used in scientific discourse about theories and models. One approach to meeting these requirements would be to use any common object-oriented programming language (interpreted or compiled) augmented by an extensive set of object classes which provide the high-level constructs. These classes would thereby extend the base language to turn it into a particular set of language elements. The clarity requirement suggests that a simple interpreted language (e.g. Python) would be more appropriate than a compiled language such as C++ or Java. But the need for the language framework to support a variety of software components, including databases and user interfaces, provides more serious constraints, and, it turns out, suggests a different approach which yields rich rewards in terms of clarity and modularity.

We would like these languages to be amenable to storage in databases and to support queries over the constituent elements and structures in a model and its annotations. For this to be possible, the model description cannot contain any imperative programming; it must be purely declarative. The reason for this is that it is impractical to do searches over imperative programs without running the programs under all possible sets of boundary conditions, something that is clearly infeasible. Declarative forms are also closest to the way in which people discourse about models. Consider the following passage from a textbook describing a neural structure:

> 'This simplified model of the CA3 pyramidal cell has 19 compartments with a soma in the center and two linear chains of compartments to represent the apical dendrites and the basal dendrites ... [T]his cell has fast sodium, delayed rectifier potassium, high threshold calcium, transient potassium and calcium activated potassium conductances.' (Bower & Beeman 1998, p. 130.)

This passage exemplifies the terms commonly used in descriptions of neural systems: 'soma', 'dendrite', 'conductance', etc. These are the kinds of constructs that model-description languages should provide.

Our ideal, then, would be to have model-description language elements which are entirely declarative in nature. This is possible, in principle, if the language is extensible: simply require that any aspect of the model description that requires iterative or sequential behaviour be hidden behind a name. For example, the phrase 'linear chains of compartments' in the quote above specifies a very particular structure which requires a loop to implement in the general case, but which can be referred to with the simple naming used in the quote.

### (c) *Replacing the programming*

Purely declarative descriptions are possible when every structure in the model can be named (e.g. 'parallel projection') and therefore codified. But unfortunately, the essence of research involves developing new constructs that may be ill-defined, in flux, badly parameterized, etc. It is precisely as these new, hypothesized, prototype concepts that are under investigation become better understood and accepted that they can be reified with a name and formal parameterization. We can be sure that any model description language which requires all structures to be available in the language *a priori* will not be used for long, if at all, by the research community.

How can we retain the advantages of the declarative approach yet not be restricted to an ossified language? An answer is suggested by considering how we will deal with existing constructs. There is already in existence a wealth of established structures in the neuroscience modelling community which we need to make available in the model-description language (e.g. 'dual-exponential channel'). These existing structures can only be made available by providing some computational module that implements them. Such a module has a well-defined interface to the model description language, exposing the parameters which describe the structure. If we make module-creation facilities available to the modeller, then the language can be augmented and can evolve as research progresses.

Our answer, then, is to make available to the modeller methods for referencing encapsulated imperative code fragments (a *component*) from a model description. The interface to the code fragment presented to the model-description language is purely declarative. We provide the means to program these components in a variety of languages (see § 5(c) for details). We envisage script languages being used for small, in-flux structures. As a structure becomes more stable (e.g. its parameterization settles down), we envisage that it will be recoded in compiled languages for greater efficiency. Many research models have structures which undergo rapid change for a period and then become stable as the focus moves to other aspects in the model (or indeed to experimental work).

It may seem unlikely that anyone would recode a model component from (say) a scripting language to a compiled language, once the component is functioning satisfactorily. However, there are incentives for doing this. For example, the computational efficiency of a compiled version is usually much greater. Moreover, the compiled form provides a certain amount of control over distribution of intellectual property: modellers may choose to distribute compiled components publicly without giving away the hard work that went into creating the actual program code. Further, journals in the future may require that models used in a paper be made publicly available, at least so that other researchers may duplicate simulation results. (There are already requirements to publish data in a number of research programs such as the Human Genome Project.) In any case, if a model is to be useful outside the research group in which it was developed, it must be documented adequately, and this will require much greater effort than that involved in recoding a small piece of encapsulated script code in a compiled language.

### (d) *Non-textual model descriptions*

It is becoming increasingly common in scientific simulation and programming environments to provide graphical programming tools that allow the modeller to describe a model in terms of graphical objects. For example, XNBC (Vibert 2001) and NEURON (Hines & Carnevale 1997) provide graphical methods for constructing neurons and networks of neurons. There is also increasing interest in basing models on experimental data directly rather than explicitly specifying the neural structures using interpreted data. For example, translators for common morphology formats have been developed for simulation environments such as NEURON and GENESIS. This kind of capability requires code components that can read in the data files and convert them to appropriate forms.

Both of these tendencies are examples of the trend away from textual model descriptions at the user level. Now the modeller interacts primarily with the non-textual tools, and it is the tools that then create and manipulate the descriptions. The implication of this trend is that while a textual description is necessary (modellers may still use it as a last resort if the graphical tools are inadequate in some situation), the requirements for the description language are somewhat changed, although as we shall see the result is not much different. The most important new requirement is that the model description language be able to support the non-textual tools. That is, the non-textual tools must be able to read and use effectively models described in the model description language, and also to write out such descriptions. This requires that the level of conceptual control (Hines & Carnevale 1997) presented to the modeller in the non-textual tool must be mirrored in the model-description language. For example, if a graphical network builder has the concept of a 'population' of cells, then the model-description language should also have this concept.

Fortunately, we have already noted that, for clarity, model-description languages should include constructs and conceptualizations which are at the level of the scientific discourse—and this is the origin of the concepts used in non-textual tools (e.g. the 'population' concept referred to above). Thus, the addition of support for non-textual tools is not onerous. In fact, the additional information that the language may need to accommodate will be more concerned with presentation; for example, additional hints may need to be encoded for layout on a canvas or in a virtual world for visualization and visual specification tools.

## 5. TEMPLATES FOR NeuroML

The language elements we have discussed, which are focused on providing relevant conceptual structures for particular levels of modelling, can provide the connection to databases of models. The language elements generate a stream of text to describe any particular model. If we are to store the model in a database, we need a formal characterization of the structure of the stream of text. The language-element specification is such a characterization. Another equivalent characterization arises if we treat the stream of text as a stream of data. In database parlance, what is then needed is a 'data model' to describe the structure of this stream. The language elements are a form of data model, but another equivalent form is that given by the the traditional database 'schema'. These in turn are equivalent to what object-oriented programmers refer to as a 'class'. To avoid confusion between the terms 'data-model' and 'model description' and to make clear that there is equivalence between the concepts of data model, schema and object class, we refer to them collectively with the term *template*.

Our prescription that the language elements be equivalent to database schemas and object-oriented classes

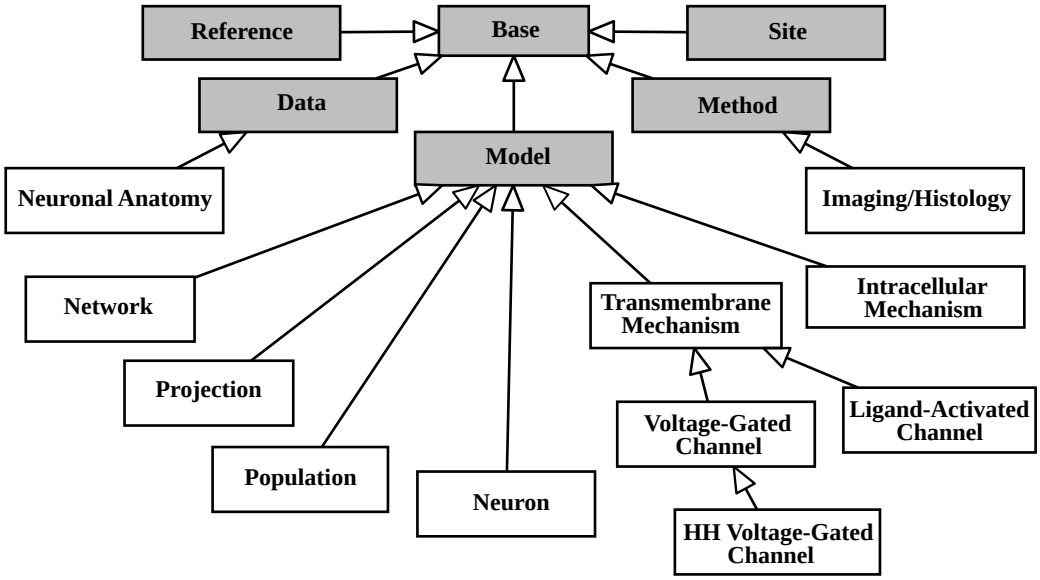


Figure 2. All templates are derived from **Base** or another existing template. Open arrows indicate inheritance, pointing from inheritors to their parents. The five first-level templates (i.e. **Reference**, **Data**, **Model**, **Method**, **Site**) are taken from the scheme developed by Gardner *et al*. (1999, 2001*a*). The additional templates derived from these five constitute the elements of NeuroML necessary to represent some common types of neural structures ranging from networks of neurons to cell membrane mechanisms. The diagrammatic notation we use is based on UML (Eriksson & Penker 1998; Oestereich 1999), an industry standard visual language for specifying software systems. We explain our notation further as needed in later figures.

places restrictions on the languages, most notably that there is no execution semantics. In other words, the language elements cannot contain conditional statements, iteration or the like. But these are exactly the restrictions we want in requiring our language elements to be declarative. There is a happy confluence between the requirements for language clarity, database information structure and programming elegance.

In the remainder of the paper we shall describe in more detail the particular conceptual structures described by the language elements at several levels, using the template terminology. A template defines how a particular kind of conceptual structure in the language element (or database object in a database, or class in an object-oriented system) is to be expressed, meaning the structure of the concept's representation, the attributes or slots in the representation, and the types of data values that are permissible for each slot. In other words, a template defines a model-description language element. Different object templates are used to create objects that represent different kinds of information. As explained below, we propose to organize templates in a simple hierarchy, and we provide facilities for defining new templates. NeuroML is the XML-based language defined by this template hierarchy; we use the term NeuroML to refer both to these templates and their specific XML instantiation.

### (a) *Template hierarchy*

The template hierarchy we present here is intended as a provisional structure for NeuroML. As mentioned above, our current efforts in NeuroML have been limited to structurally realistic models of neural systems spanning the scales from networks of neurons to cell membrane mechanisms. In collaboration with others in the neuroscience modelling community, we expect to expand this in both coverage of other scales and more in-depth coverage of the phenomena we have so far examined. This is a large task and we draw on the work of others who have focused on other aspects of this enterprise. In particular, we have used the pioneering work of Gardner *et al*. (1999, 2001*a*) in the application of mark-up languages to neuroscience, and use most of their schema definitions for NeuroML's first-level basic templates. We hope by this to ensure compatibility with their efforts at developing a common database exchange language.

Our modelling component of NeuroML is specifically geared towards computational models: the elements of the language are not merely descriptive, they are defined with careful attention to how a simulation environment can use them to produce executable models. Although the top-level, generic constructs in the language are closely aligned with those of Gardner *et al*. (1999, 2001*a*), the detailed constructs differ (in particular in the templates for describing models) simply because the goals of the efforts are different. We expect that the Common Data Model being developed by Gardner *et al*. can be used to encapsulate descriptions *about* models, such as which data-sets it accounts for. The modelling component of NeuroML, in contrast, is used to describe the *conceptualizations* underlying models, such as the types of projections in a network or the abstraction used to represent the complexity of neuronal morphology.

The representational framework that we use is based loosely on object-oriented programming concepts. One of the key ideas used is the concept of inheritance applied to templates. All templates are derived from either a specific one called **Base** or an existing template; **Base** is therefore the root of a template hierarchy, illustrated in figure 2.

The **Base** template itself contains few attributes, in the expectation that it will be subclassed to represent objects. Templates derived from **Base** then add successively more detail, starting with general-purpose constructs and on through increasingly more refined and specialized elements at the leaves of the hierarchy tree. The definitions of the templates are presented in § 6.

Each template inherits the same attributes as the template from which it is derived, and in addition, may add its own set of attributes. We impose the limitation that derived templates may only add new attributes and not delete existing ones. For example, if a given template has attributes x, y and z, a derived template can only add other attributes; it cannot change or delete x, y or z. We impose this restriction to make search operations more manageable: if existing template attributes were allowed to be modified, there could be no assurance that the fields retained their meanings when moving from higher levels in the hierarchy to lower, more detailed ones, making the kinds of search operations described below impossible.

The object-oriented style of representation is useful for a variety of reasons. First, the existence of categorical templates allows user-interface tools to present the user with intelligent search forms. Specifically, we envision that a search interface may prompt the user to specify the type of object to search for (which is equivalent to specifying the template), and based on the user's choice, the system may construct a fill-in-the-blanks form using knowledge of the attributes defined by the template. Gardner *et al*. (1999, 2001*a*) have demonstrated the utility and feasibility of this approach.

A second reason is that, by choosing the search category appropriately, searches can be made more or less specific. Because of the hierarchical relationships between templates, a user can select a template in the middle levels of the hierarchy, and search operations can be designed to encompass all objects that are below it in the hierarchy. This means, for example, that a search using **Model** will encompass objects created from templates derived from it, such as **Neuron** class objects, **TransmembraneMechanism** class objects, etc.

A final reason for the utility of the representational framework presented here is that software can be made modular and extensible. New software modules can be developed alongside new templates, customizing the system to interact with new types of objects without redesigning or restructuring the whole system. For each representation derived from an existing template, all the software elements that worked with the parent template will also work with the derived templates. This is because the derived template can only add attributes, and while the existing tools will ignore the new attributes, they will continue to work with the attributes that were inherited from the parent template. Developers can write new software modules that interact with the additional fields in the new templates and these software modules can be loaded on demand, extending the software's functionality.

### (b) *XML-based encoding of templates and models*

Extensible Markup Language (XML) (Bosak & Bray 1999; Bray *et al*. 2000) is a language used to express self-describing, semi-structured representations of information. It provides a way of marking up data with semantic tags that describe and structure the contents of the data. Although XML is typically thought of as a document format similar to HTML, in fact it is more general. It is a notation, a 'metalanguage', a way of organizing a stream of data and marking up the different parts so that a program can parse the stream into constituents. In the words of one of its chief architects, 'Just as HTML created a way for every computer user to read Internet documents, XML makes it possible, despite the Babel of incompatible computer systems, to create an Esperanto that all can read and write. Unlike most computer data formats, XML markup also makes sense to humans, because it consists of nothing more than ordinary text.' (Bosak & Bray 1999).

We use XML schemas (Biron & Malhotra 2000; Fallside 2000; Thompson *et al*. 2000) to describe model templates, and actual models are encoded in NeuroML using these schemas. All the constructs described in § 6(a–d) have direct equivalents in an XML schema. An XML schema specifies the structures that are syntactically correct in a schema-compliant stream of data, enabling syntax-checking in software. The NeuroML schemas ensure that model descriptions in NeuroML adhere to the structural requirements of the model templates. The schemas also support other software tools such as visual editors which can use them to generate syntactically correct NeuroML model descriptions.

For handling representations in a diverse collection of databases, simulation environments, and other software tools, XML offers three key features: (i) Separation of the template description (in the form of an XML schema) from the content. Provided there is agreement on basic guidelines and protocols, databases, simulation environments, etc., can have their own specialized representations. If the schemas are advertized, we can retrieve them separately and use them to determine which model classes a particular software tool uses in common with other tools, and which classes are unique. (ii) As mentioned above, schema descriptions can be used to dynamically create user interfaces, for example, for search forms or visualizations. XML makes this easier to implement because its format lends itself to simple interpretation and manipulation by tools. (iii) The text-based format of XML is a powerful way by which to handle novel data types: a database server that contains specialized data can provide, say, JavaScript code implementing an applet for viewing the data. The applet script can be sent as part of the XML document when the document is retrieved by a client program.

### (c) *Incorporating programmed structures and scripts using XML schemas*

An important issue is how the descriptions can be augmented to incorporate model structures best described in imperative terms (i.e. using a programming language). Our approach to this problem is to use XML schemas to define new templates which refer to dynamically-loaded code modules. Figure 3 illustrates the concept. The XML schema file `NeuroML.xsd` defines the standard set of templates available for use in a model, such as 'neuron', 'network', etc. Commonly used extensions are defined in

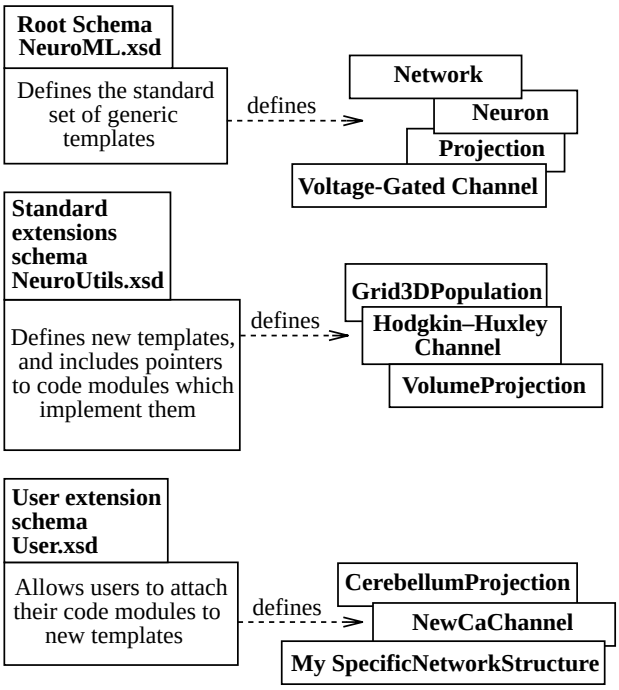


Figure 3. Overview of the organization of XML schemas in this approach.

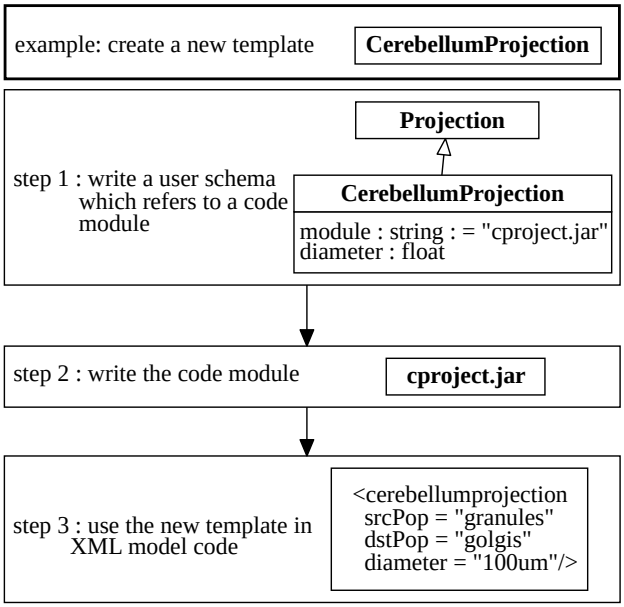


Figure 4. The steps involved in adding a new user template with an associated code module and using it. First, the template describing the interface to the code module is defined. Second, the code module is written. Third, the new template is used in a model description.

```
<generalprojection <srcPop = "granules" destPop = "golgis" seed = "123">
    <connectionmethod>
      <!--A section of script code embedded in the XML file -->
      if (Random.sample() < 0.1)
        { makeConnection(); }
    </connectionmethod>
</generalprojection>
```

Figure 5. In some cases, it can be useful to embed short sections of script language code into a model description. This example provides a user-defined connection method between two neurons; the method produces a given proportion of connections.

another schema file, `NeuroUtils.xsd`; this defines such things as volumetric projections and commonly used scatterings of populations of neurons. The program code to actually perform these functions is located in an external code module that can be loaded dynamically by a simulation environment, and the schema document refers to this code module. User-defined extensions are specified in the same way. Such a code module can be implemented in a variety of forms such as a JavaBean or shared object library; what is important is that *a new label is defined for the new capability, and models are defined declaratively in terms of these labels*—the code is not directly part of the model. To put this in specific XML terms, we can define new labels such as `<volumeprojection>` and `<cerebellumprojection>` which can be used in defining a model; the actual code to implement a 'volume projection' (which might have to make connections based on distances between neurons, sample some probability distributions, etc.) is part of the specification of what `<volumeprojection>` means in the language.

Defining user schemas for new facilities provides information that can be used by schema-aware parsers and editors, allowing for such things as automated data type checking. The cost of moving code to the schema is that more steps are involved than embedding code directly into a model (see figure 4 for an example). However, we believe the advantages gained by not coding but declaring models outweigh this; in any event, much of the work can be automated by tools.

Giving users the ability to define new templates is very appealing. But for many modelling tasks, it may be simpler to use in-line scripting code. The example in figure 5 shows how a small section of scripting code could be introduced to implement a user-defined projection scheme between two populations. The user code in the figure samples a random distribution in order to decide whether to make a connection between two neurons; in general, such a decision can be highly model specific, for example calculating the proximity of an axon to a dendrite. In such cases the code would be more complex than this.

### (d) *Semi-controlled vocabularies*

Many of the attributes in the templates have essentially unconstrained values. For example, there are no *a priori* constraints applicable for such things as the name of a model. But in many other cases it is useful to place constraints on the permissible values of an attribute, especially string attributes, in the form of a controlled vocabulary. Gardner *et al*. (2001*a*,*b*) are leading an effort to define a set of controlled vocabularies for use in biological databases. These are meant to capture the allowed values for an attribute. A controlled vocabulary is especially useful for search operations: when database objects use attribute values drawn from a common vocabulary, it is much more likely that a database search will succeed in finding a match than if users are given free reign to enter

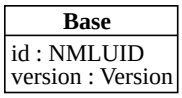


Figure 6. The **Base** template. In this UML-based notation, the name of the template is shown in bold face at the top, and its fields/attributes are listed below it. The name of a field is to the left of the colon; its data type is to the right.

any value for every attribute. Controlled vocabularies reduce the chances that differences in spelling, the use of different terms having the same meaning, and other incidental effects will cause a search to fail.

We borrow this general idea, but in some situations we also allow for attribute values that are not strictly limited to values from a set vocabulary. We call this a semi-controlled vocabulary. Such *semi-controlled vocabularies* can be attached to an attribute to provide a set of initial, suggested values in the user interface of an editing tool. When a vocabulary is available for a given attribute in a model template, the editing field for that attribute can provide a pull-down list containing the set of suggested values, allowing the user to easily select a value from the list. If none of the available values is suitable, the user may enter a different one.

Vocabularies can be easily defined in XML Schemas. The use of a semi-controlled vocabulary involves a trade-off as compared with a strict controlled vocabulary. The provision for allowing users to type in new values means that search operations may no longer be as effective. However, we feel that we are not able to define a sufficiently comprehensive controlled vocabulary for all attributes, and that, moreover, users would react negatively to an interface that does not allow them to type in new attribute values when the predefined set is insufficient. We believe that we can obtain most of the benefits of a controlled-vocabulary approach by providing reasonably comprehensive default vocabularies, so that users are likely to find a predefined value close enough for their needs.

## 6. THE TEMPLATES

In this section, we describe briefly our current templates at the top level of the hierarchy and under the **Model** template. A detailed definition and explanation of one of the templates is given as an example in Appendix A. However, for up-to-date information on templates used in NeuroML, we urge the reader to access the NeuroML web site at http://www.neuroml.org.

### (a) *The basic templates*

We begin by summarizing the preliminary versions of the **Base** template and the five templates derived immediately from it in the hierarchy shown in figure 2: **Reference**, **Method**, **Model**, **Data**, and **Site**. These are general data structures that are not limited to representing specific biological objects such as neurons and ion channels. The more specialized templates that we have developed for representing models of neuronal networks, single neurons and associated elements such as ion channels, are presented in § 6(b–d).

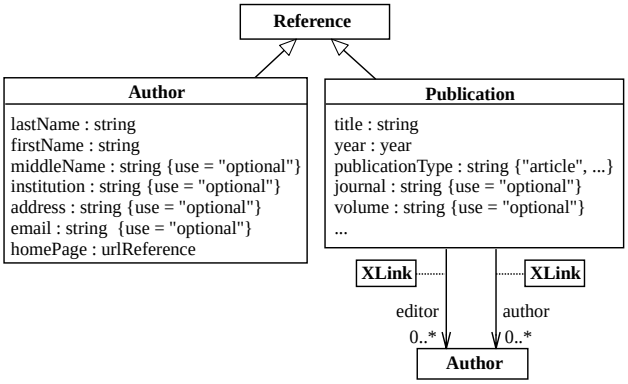


Figure 7. The **Reference** template and two derived templates, **Author** and **Publication**. Text enclosed in braces next to attribute types (e.g. {use = ''optional''}) indicates constraints on the possible attribute values; we use XML Schema language to express constraints because we are primarily interested in the XML encodings of templates. The XLink notation indicates an association with a separate instance of an object of the associated type (in this case, **Author**). The type XLink signifies the use of the *XML linking language*, XLink (DeRose *et al*. 2000). The numbers on the arrows indicate the number of allowable instances of this attribute; e.g. '0 .. *' means 'from zero to unboundedly many', and the labels on the arrows indicate the names of the attributes storing the associations.

(i) *The* **Base** *template*

The **Base** template, shown in figure 6, has only two attributes, id and version. The former places a unique identifier on every object; the latter allows the system to track the evolution of data objects. The id attribute of type NMLUID is an identifier that is unique to a given model within a simulator or database. The type Version contains fields for such things as a time-stamp, a version number, the version number of the immediate parent version, etc.

Since all objects must be derived from **Base**, all objects inherit id and version attributes. This may at first seem odd, because it may seem that some types of objects such as 'data' do not need versions. However, we believe that all database object representations can benefit from having version information, because it enables changes (such as the addition of more details) to be tracked through version control facilities that may be implemented in editing tools. This will be especially useful when objects are transmitted between users' personal databases: it will allow users to determine whether their local copy of an object is the same as or different from one retrieved from a remote database.

(ii) *The* **Reference**, **Author** *and* **Publication** *templates*

The **Reference** template serves as a starting point for references to such things as authors and publications. The definition is shown in figure 7. **Reference** inherits the id and version attributes from the **Base** but adds no new attributes; both **Author** and **Publication** inherit id and version as well, and each adds other attributes specific to its role. (The attributes implicitly inherited from the **Base** template are not repeated in figure 7, following UML convention.) This general scheme follows the ideas put forward by Gardner *et al*. (1999, 2001*a*).

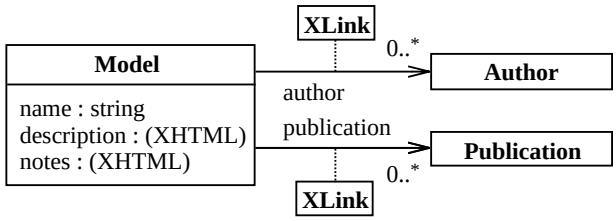


Figure 8. The **Model** template.

The **Author** template adds attributes for identifying a person by name, address, web home page and other characteristics, as shown in the figure. The **Publication** template shown in the figure adds attributes describing literature references. The attributes in the template are based on BibTeX records (Lamport 1994), some of which are shown in figure 7. In a well-designed user interface, the choice of publicationType made by the user can trigger the interface to selectively present only those attributes relevant to the type of reference in question.

(iii) *The* **Data**, **Method** *and* **Site** *templates*

The **Data** template is another subclass of **Base**. It provides basic support for storing data in a database or pointing to data stored in a remote database. Objects of this type are pointed to from other templates such as **Model** (see §6(a)(iv)). The **Method** template is intended to capture information about experimental methodologies. **Data** and **Method** are based on the DATA_ELEMENT and METHOD_ELEMENT structures, respectively, developed by Gardner *et al*. (1999, 2001*a*). Their full definitions are available from the NeuroML web site.

The **Site** template is intended to capture information about such things as neuronal recording sites, brain regions, etc. We have not defined these templates but anticipate that we will draw heavily on the work of the NeuroScholar project (Burns *et al*. 2001) and a neuroanatomy ontology project currently underway.

(iv) *The* **Model** *template*

The **Model** template shown in figure 8 is intended to serve as a common starting point for all model template definitions. It is a generic structure, not specific to any particular kind of modelling. Specific kinds of models, such as for neuronal networks (§6(b)), neural cells (§6(c)) and intracellular and transmembrane mechanisms (§6(d)), are derived by starting from **Model** and adding new attributes.

As with the other main templates, the **Model** template is an extension of the **Base** template and therefore implicitly inherits id and version attributes. **Model** then adds several more attributes. The attributes name and description allow a user to name a model and provide a brief description of it. To support the possibility of including equations, superscripts, and other formatted content, the description is stored in XHTML format. The notes attribute provides a place for recording information about a model that is not easily recorded in any other attribute, for example an output plot resulting from simulating the model. The author list is intended to point to the 'authors' of the model, and the publication list is intended to point to relevant literature, specifically articles discussing the model.

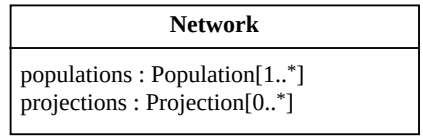


Figure 9. The **Network** template includes a number of **Neuron** populations and a number of **Projections** (possibly zero) between populations.

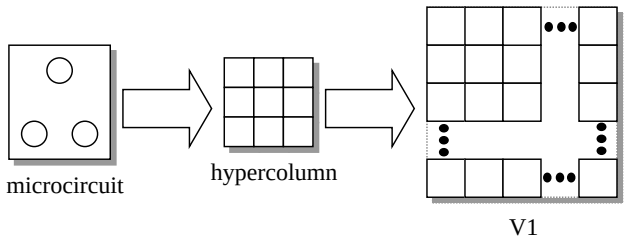


Figure 10. A set of hierarchical populations: V1 is composed of an array of hypercolumns, each of which is composed of a small array of microcircuits.

(**b**) ***Templates for representing networks of neurons***

The main template for representing models of neural circuits is **Network**. It makes use of several other templates, in particular **Population**, **Projection** and **Neuron**. Figure 2 depicts the hierarchical relationships between them. In this section, we discuss **Network**, **Population** and **Projection** and associated templates. We leave the neuron and cell mechanism templates to §6(c) and (d).

The **Network** template (figure 9) comprises populations of cells connected by projections between them. (In the degenerate case, a population can be a single cell and a projection a single synaptic contact.) Both populations and projections are compositional—they can be composed of other populations and projections, respectively. Populations are named, structured sets of cells. One population can be a subset of another, or composed of a union or disjunction of other populations. Figure 10 illustrates a three-level hierarchy of populations.

'Projection' is a complex concept. An informal description of the projection construct is that it maps one population having a particular type of structure (not necessarily spatial) to another population having another type of structure. Thus, it is essentially a mapping from one structure to another. The population structure concept therefore precedes the projection concept, i.e. we cannot define projections without first having defined population structures. Furthermore, a given population may need to be associated with more than one structure. One projection may most naturally view the population as having a particular structure, while another projection most naturally views it as having a different structure. Yet another structure may be more appropriate for visualization purposes. Therefore we need to allow the association of a variety of structures with a particular population. We allow for a default population structure in a projection if none is specified. In many models there will be only a single way in which the structure of a population is

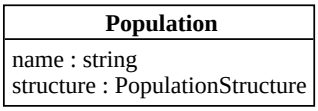


Figure 11. The **Population** template.

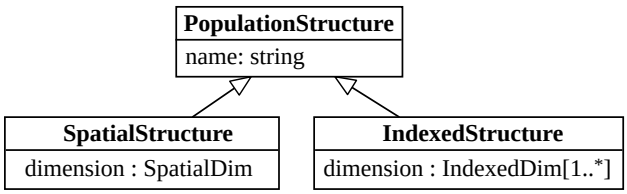


Figure 12. The **PopulationStructure** template.

conceptualized (e.g. a 2D array), so we enable a single point of specification, i.e. associated with the population rather than the projection.

Geometrical structures are the type most obviously applicable to neural models, but there may be other types. In a geometrical structure there will usually be one or more dimensions in some coordinate system, such as Cartesian, polar, etc. We can generalize a little to some non-geometrical structures which can be viewed as geometrical by allowing some or all of the $N$ dimensions to be indexed by enumerated labels rather than numeric indices. This is formally equivalent to the numerically-indexed system but may be more natural for the modeller. Initially we restrict the templates to structures characterized by a number of dimensions and a well-formed coordinate system, with the extension that the indices may be derived from an enumerated type other than integer.

(i) *Populations*

The provisional **Population** template is shown in figure 11. The attributes common to all populations are its name, and the default structure associated with the population. We expect to provide subtemplates for populations of neurons, and populations of networks. We also expect to provide templates for 'Views', which are pseudo-populations derived from existing populations (e.g. subset, union) with perhaps a different associated structure.

Structuring of populations is provided by the **PopulationStructure** template, which encompasses all types of structuring of populations (figure 12), with one attribute encoding its name. We have defined two types of structures at this time: **SpatialStructure** and **IndexedStructure**. **SpatialStructure** represents a structure which is layed out in a continuous metrical space, e.g. 3D real-world space; spatial structures have a number of linear or polar dimensions. **IndexedStructure** represents structures with no associated continuous metrical space but in which the elements of the population are accessed via integer indices, e.g. a 2D array of neurons; indexed structures have a number of linear or enumerated dimensions.

These templates and structures allow us to create hierarchical, structured populations. We anticipate that, in the great majority of models developed in the near future, if hierarchies are used at all they will be very shallow.

(ii) *Projections*

Figure 13 illustrates a three-level hierarchy of projections. The description of the connections at the level of V1 is given in terms of a projection between the hypercolumns of V1 (nearest-neighbour), where the projection between the components (microcircuits) of any pair of linked hypercolumns is one-to-one, and the projection between the components (neurons) of any pair of microcircuits is the two axonal connections shown. The critical feature here is that a projection defines a structured set of links between components of one or two populations, where the structure of each individual link in the set can be another projection at a lower level.

A projection specifies a mapping between a source population and a destination population. The **Projection** template (figure 14) has two key attributes: (1) structure, which specifies the mapping between the populations, and (2) linkType, which describes the projection from an element of the source population to an element of the

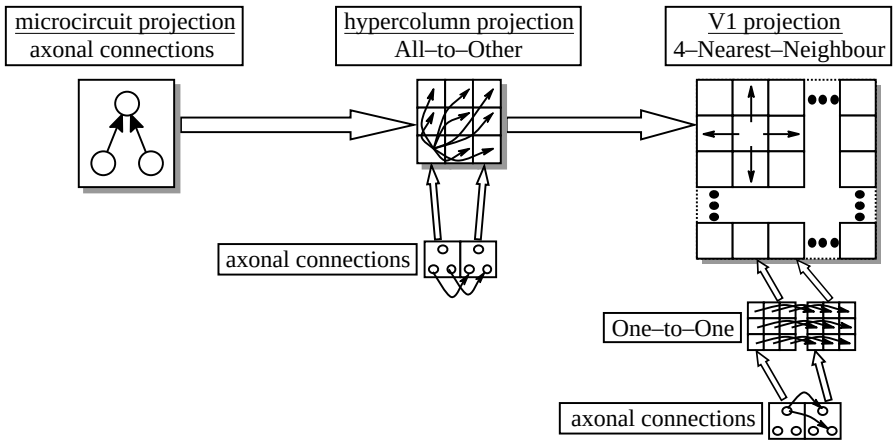


Figure 13. A set of hierarchical populations: the microcircuit is described with simple axonal connections; the hypercolumn is described as an all-to-other projection between microcircuits, where each link in the projection is the bundle of axonal connections shown below; V1 is described as a four-nearest-neighbour projection between hypercolumns, where each link in the projection is another projection, one-to-one, between microcircuits in the hypercolumns, and each link in the one-to-one projection is the bundle of axonal connections shown below.

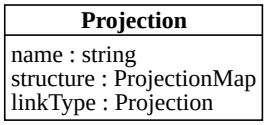


Figure 14. The **Projection** template.

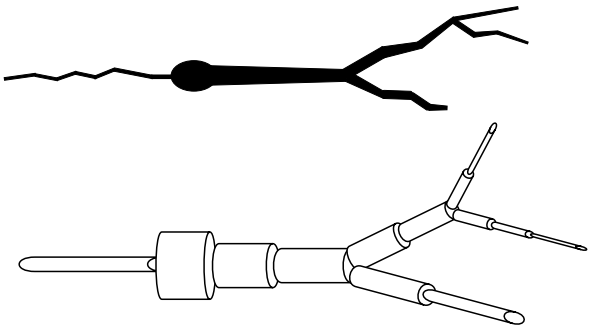

Figure 15. Stylized illustration of a neural cell reduced to a finite approximation constructed out of cylindrical segments.

destination population. In a population of cells, this will be a synaptic connection. In a population of populations, it will be a web of synaptic connections (i.e. another projection), as illustrated above.

This structure allows us to define hierarchical projections, such as an all-to-one mapping between cortical columns in two cortical areas in which the projection from one column to another is a complex of synaptic connections. It also allows multiple levels of hierarchy, for instance a one-to-all mapping between a set of cortical areas.

The bulk of the work is left to the **ProjectionMap** template (not shown). At this point we are leaving the realms of well-understood and agreed-upon structures in neuroscience. The experimental data suggest a great diversity in the structure of projections between neuron populations (see, for example, Burns 2001). The modelling literature uses a variety of projection structures, increasingly diverse the closer the model is to the experimental data. Until well-understood and accepted projection structures emerge from experimental and theoretical work, we provide simple geometric structures, and the ability to program structures using a programming language via the code-module facility described in § 5(c).

We expect that, over time, some of these programmed projection structures will become widely used with well-accepted parameterizations. Then we expect the interface to these structures to become standardized and encoded as a template. The advantage of template encoding is that aspects of the projection structure become accessible to other NeuroML-aware software components such as database query engines or visualization tools.

### (c) *Templates for representing models of neural cells*

The **Neuron** template extends the basic **Model** template with additional attributes for describing a neural cell. Our efforts so far have been directed at structurally realistic models, rather than abstract models such as integrate-and-fire neurons; however, we expect that in the future, we will define templates for describing abstract neurons as well.

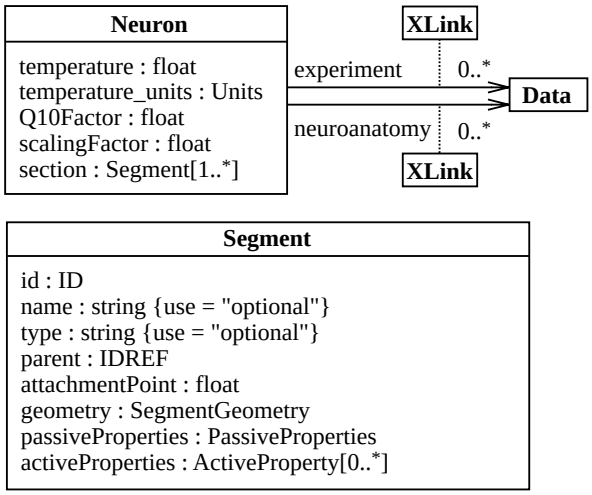


Figure 16. The **Neuron** template.

Most contemporary approaches for creating realistic models of neural cells are based on cable theory and compartmental modelling (Hines 1984; Segev & Burke 1998; Rall & Agmon-Snir 1998). This involves taking dendrites, axons and cell bodies, and modelling them as a number of segments (typically cylindrical in shape, but sometimes spherical or conical), as shown in figure 15. In software simulation systems such as NEURON and GENESIS, these segments may then be further subdivided into *compartments*, which are finite-difference approximations that can be defined in terms of ordinary differential equations.

The properties of a segment of passive cell membrane include such things as its capacitance and resistance; these enter into the differential equations that relate current and voltage inside and outside the segment membrane. Neural cells also have active elements such as ion channels embedded in their membranes; such channels allow ions of substances such as sodium and calcium to pass through, resulting in changes in the electrical potential across the membrane. The collective effects of a given species of ions are characterized as an active conductance, and this adds terms to the differential equations modelling a segment of a neuron membrane.

The **Neuron** template is designed to capture the structure of a neural cell model in terms of segments with embedded active conductances. The main portion of the definition is shown in figure 16. The template implicitly inherits the attributes name, description and notes from the **Model** template, and adds a number of others. The first few attributes in the top-level structure of the **Neuron** template provide the ability to specify the temperature assumed for the parameter values in the model, together with scaling factors to be used to adjust parameters for other temperatures if needed. This allows the model to be reused in a simulation at a different temperature. Attributes experiment and neuroanatomy are lists of pointers to **Data** objects that may be used to add information about experimental data related to a given neuronal cell model.

The actual structure of the cell model is described using a list of one or more Segment type data structures. Each segment in the model structure has a unique

identifier, an optional name, a pointer to a parent segment, and collections of geometry attributes, passive properties and active properties. The geometry attributes (organized in a SegmentGeometry data structure, not shown here for brevity) allow a segment to be described and orientated in three spatial dimensions. The passive properties (organized using PassiveProperties data structure) include such things as specific membrane capacitance and resistance. Information about active properties is represented using references to objects derived from either the **TransmembraneMechanism** or **IntracellularMechanism** templates described below.

### (b) *Templates for representing models of cell mechanisms*

The **TransmembraneMechanism** and **IntracellularMechanism** templates are the starting points for defining models of active properties for use in neuronal cell membrane segments. As shown in figure 2, both of these templates are derived from the generic **Model** template, and other templates are in turn derived from them.

First, a few words are in order concerning our use of the term 'channel'. As mentioned above, this term is often used to mean an individual pore through which ions may flow into and out of a cell. However, in some simulation contexts, the term has also come to be used loosely to mean an ionic conductance that models the behaviour of many thousands of individual channels (pores) as they open and close. In this usage, a 'channel' is not really an ion channel at all; it is a construct, a stand-in for a conductance that represents the massed effect of a large number of actual ion channels. It is this second sense of the term that we use here.

(i) *Transmembrane mechanisms*

**TransmembraneMechanism** inherits from **Model** and adds a few new attributes: a text string for naming the cell type from which a particular transmembrane mechanism model is drawn, and optional lists of pointers to datasets describing the experimental results from which the model was derived. The cell type attribute has a semi-controlled vocabulary associated with it. The motivation for this is to allow user interfaces (such as in the Modeler's Workspace, see § 7(b)) to present the user with a list of common alternatives. Users may select from one of the values or supply a different value of their choice.

The **VoltageGatedChannel** template inherits from the **TransmembraneMechanism** template and adds several attributes common to many models of voltage-gated channels. These attributes include the following: descriptors for the channel type and current type; the cell resting membrane potential being assumed by the channel model (this is sometimes used as part of the initialization of a simulation); the equilibrium potential of the membrane segment (the initial value of the membrane potential at which there is no net flux of the ion across the membrane); and finally, the temperature and temperature scaling factors assumed for the parameter values in the model.

Readers may wonder why the equilibrium potential is included as a user-specified value here, when in simulations it can be computed or changed by various simulated mechanisms. The reason is that this quantity is often treated as a constant in a model, or is used in the process of computing other quantities. This illustrates one of the problems with developing a general representation that can be converted into simulatable code: it is sometimes necessary to include fields that may not be fundamental quantities or may not be used in all models, but must nevertheless be included to permit the construction of those models that do need to specify their values.

The **HHVoltageGatedChannel** template is intended to represent a certain common class of ion channels used in many structurally realistic neural models. It can describe channels not only of the common Hodgkin–Huxley (Hodgkin & Huxley 1952) variety, but also a number of variants. The representation is derived partly from what is used in the GENESIS neural simulator (Bower & Beeman 1998), with additional information introduced for generality and in order to make a filled-out model more descriptive and identifiable during database searches.

The definition of **HHVoltageGatedChannel** is fairly complex. However, it is instructive because it illustrates the connection of the model representation to theoretical concepts. We therefore provide it in Appendix A, together with example portions of an XML document encoding a sample model.

## 7. IMPLEMENTATION

We are currently implementing systems to handle models defined using NeuroML. The NEOSIM Project is developing high performance tools within a modular framework to run simulation models, and the Modeler's Workspace project is developing a modular framework for interfacing simulators, editing tools and databases.

### (a) *NEOSIM*

NEOSIM is a framework to support large scale multilevel modelling of the nervous system. Its features include (i) a plug-in facility that allows different simulation components to be added for simulation, visualization and I/O; (ii) a design that enables execution on workstations together with networks of workstations and parallel machines without having to write a parallel program, and (iii) a simulation approach based on discrete event simulation (Fujimoto 1990).

Up-to-date details are available from the project web site located at http://www.neosim.org. Figure 17 illustrates the NEOSIM kernel with the separation of XML model description levels from code modules. At the time of writing, two API-identical kernels (Java for portability, C++ for high performance) are available for download. When (and if) the time comes to scale up a model to more realistic sizes, the NEOSIM kernel supports the distribution of the model onto multiprocessors, networked workstations or parallel machines without having to modify the model description.

The NEOSIM kernel provides only the basic support necessary for building and running large simulation models. All of the interesting behaviour of model components is provided by plug-in modules. These modules are intended to be developed independently by different groups, and can communicate with each other using the kernel interface. For example, a model could be built using a visualization component written in one

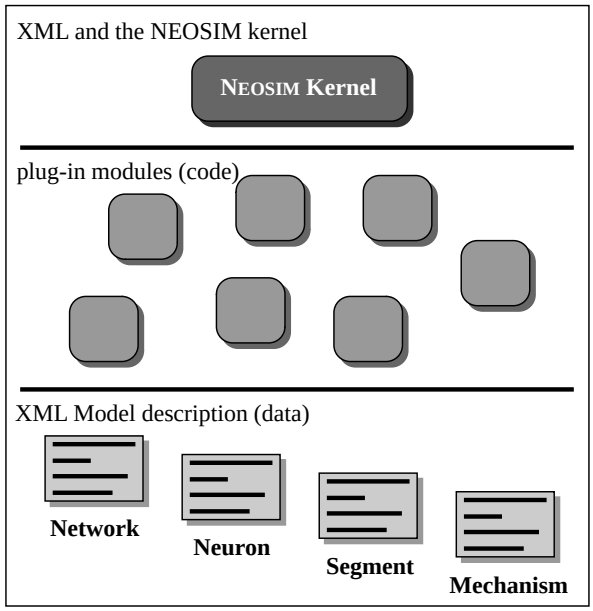


Figure 17. The NEOSIM kernel uses a number of declarative model description levels, given in different XML files. Extra *code* features are introduced using plug-in modules.

```
<neosim>
  <model>
    <network name = "demo" >
      <define–entity name = "spike"  modulename = "Spike"/>
      <population name = "GaussPop1"  entityname = "Spike"
        modulename = "GaussPop"  num = "100"
        xcentre = "0"  ycentre = "0"  zcentre = "0"
        xvariance = "10.0"  yvariance = "10.0"  zvariance = "10.0"
        xdist = "gaussian"  ydist = "gaussian"  zdist = "gaussian"
        seed = "1234"/>
    </network>
  </model>

  <experiment>
    <module name = "Spike"
      location = "http://www.neosim.org/modules/spikegen.jar" />
    <module name = "GaussPop"
      location = "http://www.neosim.org/modules/gausspop.jar" />
    <control>
      <time value = "100.0"/>
    </control>
  </experiment>
</neosim>
```

Figure 18. Sample NEOSIM XML description of a model and simulation run.

laboratory, a neuron simulation component from the NEURON simulator, another from the CATACOMB (Cannon 2001) simulator, and a number of home grown Java or C++ components. At the time of writing, a number of plug-in modules are available for downloading, including some supporting NEURON and CATACOMB models.

A clear separation between declarative model descriptions (in XML files) and plug-in code modules is supported by NEOSIM. The example in figure 18 shows an XML model description of the type currently used by NEOSIM, similar to that which will be created from the templates described above when the implementation is complete. The example shows a model which consists of a single population of entities.

The experiment section in the example contains the specifics needed to actually run a simulation: it provides the location of plug-in code modules and gives a simulation run time.

In the example, the plug-in (*GaussPop*) contains the code for scattering the members of a population across a given area, given by a centre point and variances in each dimension. The behaviour of a neuron is given in the plug-in (*spikegen*). In both cases the code is specified in a Java archive (`.jar`) file, and referenced using a URL, so that anyone with web access can run this model, and the code is downloaded automatically by NEOSIM.

### (b) *The Modeler's Workspace*

The goal of the Modeler's Workspace project (Forss *et al*. 1999; Hucka *et al*. 2001*b*) is to develop an environment for computational neuroscience providing a number of capabilities: searching remote databases for model components based on various criteria; creating new components; combining model components together and translating them into formats suitable for simulation systems such as GENESIS, NEURON and NEOSIM; managing personal databases of models and other information; and collaborating interactively with other researchers to work with models and simulations. As with NEOSIM, the Modeler's Workspace is being written in Java for portability and extensibility, using a modular architecture of the type illustrated in figure 1.

The Modeler's Workspace *User Interface* can run either as a separate application, or as an applet from within a web browser. All user interactions with the other components of the Modeler's Workspace take place through the User Interface. The *Workspace Database* acts as a private repository for a user's work (where models, notes and other objects are stored). The database contains objects that represent the different types of entities; each object is structured according to one of the templates discussed throughout this article.

Interfaces to third-party databases and also simulation packages are implemented using software plug-ins. Each kind of database must have a plug-in that mediates between the system and the database. The plug-in's task is to perform the following functions: (i) engage the network communications protocol required by a particular foreign database (e.g. CORBA/IIOP, HTTP, Z39.50); and (ii) translate back and forth between the Modeler's Workspace templates and search language and the corresponding elements of a foreign database. Similarly, interaction with simulators is also supported through plug-ins; a simulator plug-in must perform the functions of (i) interfacing to the simulation tool using Java Native Interface, network protocols, or some other means; and (ii) translating back and forth between the Modeler's Workspace representation and the format understood by the simulator. We are currently developing two simulator plug-ins, one for GENESIS and one for simulators operating in the NEOSIM framework, such as NEURON and CATACOMB. The NEOSIM interface will be particularly simple to implement because the two systems implement the same data model—the templates introduced in this article.

The existence of templates and their hierarchical organization allows the Modeler's Workspace database search

facilities to present the user with intelligent search forms as described towards the end of § 5(a). Results are shown in a tabular summary format; individual objects can be examined in more detail by double-clicking the mouse over an entry in the results table. Viewing of objects is accomplished using graphical interface plug-ins called *inspectors*. An inspector is simply a user interface module designed to let a user interact with information in a certain way. We are developing special-purpose inspectors for a variety of object types, including a neuron model inspector with a graphical, three-dimensional tree viewer/editor for working with cell morphologies, and a channel model inspector featuring graphical plots of channel characteristics.

## 8. DISCUSSION

The work described here has been motivated by the increasingly large-scale modelling needs in neuroscience. Structuring the exploding volume of experimental data according to the models which use it is likely to lead to greater synergy between modelling and experimental efforts. It is notable that the NeuroScholar system (Burns 2001), has complementary aims. The increasing scale and complexity of models, especially those crossing levels of inquiry, require model description capabilities which are as declarative as possible, for reasons of clarity, portability and modularity. We have shown that a single model description framework, the templates described above, can support both database information structure and simulation model execution. The inclusion of an ability to interface imperative code modules to the declarative templates via a well-defined, parameterized interface, provides a path for speculative research-oriented conceptualizations to be supported and, crucially, to gradually migrate into the model description language (the templates). Thus, we believe that this form of model description is, to a reasonable extent, future-proof. For both database entries and simulation execution, the model description capabilities are naturally extensible to support evolution of conceptualization of brain function.

Our major aim in this work is to facilitate collaborative modelling work which requires the ability to share and reuse model components developed at other times in other laboratories on other computational systems. We believe that the underlying software technology (Java for execution, XML for description) can now support these requirements. The motivation for collaborative work is twofold. First, as simulation models become more complex and cross spatio-temporal levels of inquiry, it is likely that the expertise required to model all aspects of the system adequately will not be present in one laboratory and perhaps not at one time. Second, development of essential scientific tools such as NEOSIM and Modeler's Workspace is best done in a highly distributed, collaborative manner. No one laboratory has the scientific expertise across all the levels we need to support, and no one laboratory has the funding or software engineering capability to produce these tools *in toto*. Thus, a component based approach is essential both for the scientific aspects of the work and the software methodologies which support the science. The essence of component based approaches is architecture. In this paper, we have described an underlying information architecture for one part of these systems, that concerned with describing the structure of computational models.

We thank Dan Gardner, Robert Cannon, Marc-Oliver Gewaltig, Greg Hood, Paul Rogister, Michael Hines, Gully Burns and Henry Thompson for helpful discussions about the structure of NeuroML and this paper. This work was supported by National Institutes of Health Human Brain Project grants MH57358 and NF00002, and an Academic Equipment Grant (EDUD-7824-000127-US) from Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303.

## APPENDIX A: DETAILED TEMPLATE DEFINITION FOR VOLTAGE-GATED ION CHANNELS

We present here our current definition of one of the templates discussed in Section 6(d)(i). At the time of writing, the representation is still evolving; the latest template definitions are available from the NeuroML Web site, http://www.neuroml.org.

The diagram for the **HHVoltageGatedChannel** template in figure A1 shows that the top-level data structure consists of a number of *ions*. A given channel may pass more than one kind of ion, although most channel models involve only one ion. Though it may arise infrequently, the case of multiple ions must be supported, so the ion attribute in **VoltageGatedChannel** is a list of Ion data structures. As shown in figure A1, each Ion structure contains the following attributes: name, a string that records the user-chosen name of the ion (e.g. 'potassium'), and gatingVariable, a list of one to three GatingVariable types discussed below.

In neural modelling, individual ionic conductances are often described in terms of a small number of gating variables that control the flow of ions through the channel. The GatingVariable data structure contains attributes related to the properties of a gate. The collection of gating variables in a particular channel model captures the bulk of the information necessary to describe the behaviour of the channel. Each GatingVariable structure contains the following attributes:

name: The name of the gating variable, constrained to be one of the strings 'X', 'Y', or 'Z', following a naming convention used in GENESIS. These refer to the variables in the equation for channel conductance,

$$G_k = \bar{g}_k X^{X_{\text{power}}} \Upsilon^{\Upsilon_{\text{power}}} \zeta^{\zeta_{\text{power}}}. \tag{A1}$$

In this equation, $G_k$ signifies the conductance due to a channel of type $k$, and $\bar{g}_k$ is a normalization constant that determines the maximum possible conductance when all the channels are open. The variable $X$ corresponds to the gating variable usually called $m$, and $\Upsilon$ corresponds to the gating variable usually called $h$. To give an example, for the case of the sodium channel in the squid axon, $X \equiv m$, $X_{\text{power}} = 3$, $\Upsilon \equiv h$, $\Upsilon_{\text{power}} = 1$, and $\zeta_{\text{power}} = 0$, to produce the common Hodgkin–Huxley formula $G_k = \bar{g} m^3 h$.

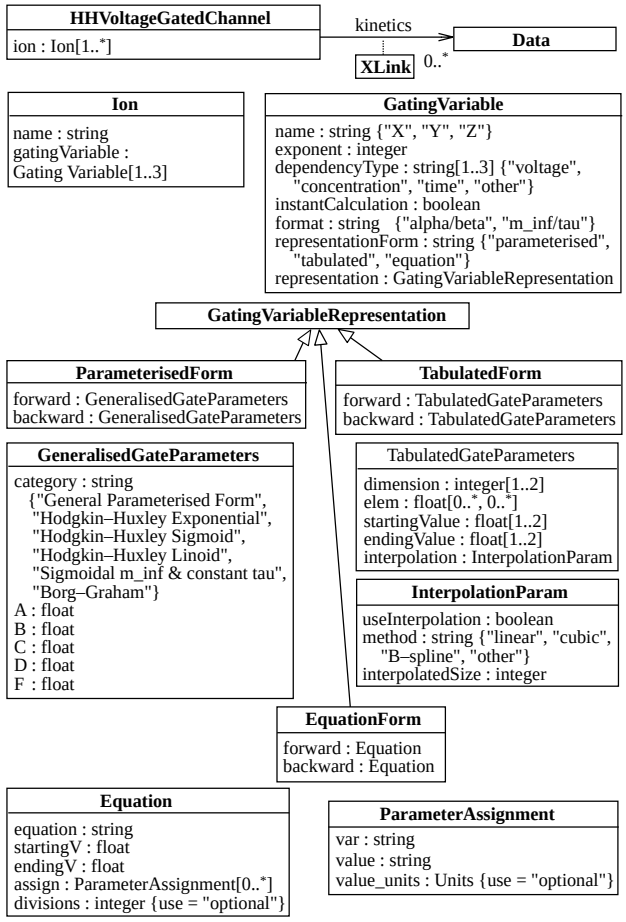


Figure A1. The definition of the **HHVoltageGated Channel** template.

exponent: An integer indicating the value of $X_{\text{power}}$, $Y_{\text{power}}$ or $Z_{\text{power}}$, as appropriate.

dependencyType: A list of strings that describes whether a given gating variable is dependent on voltage, concentration, time, or other quality. It is a list because a gating variable can be dependent on more than one quality simultaneously.

instantCalculation: A boolean variable used to note whether a gating variable's value is meant to be determined instantaneously from $\alpha/\beta$ or $m_\infty/\tau$ values, rather than solving differential equations. If true, the value of a variable $X$ is determined using $X = \alpha/(\alpha + \beta)$. This is often used with a Z gate to model a multiplicative factor in a calcium-dependent conductance, or to implement Morris–Lecar models having zero time constant for activation (Bower & Beeman 1998).

format: Whether the data describing the gating variable's behaviour are stored in Hodgkin–Huxley-style $\alpha/\beta$ form, or state variable $m_\infty/\tau$ form (Nelson & Rinzel 1998). Some users prefer one format over the other—often, modellers tend to prefer $\alpha/\beta$, and experimenters like to show $m_\infty/\tau$ because they are more closely related to the results of voltage clamp experiments. The choice is made by the user and should reflect the format in which the user actually input the parameter values. The data are left untouched regardless of the user's choice; it is up to the simulator program (or other consumer of models in a database) to do whatever is necessary to translate between $\alpha/\beta$ and $m_\infty/\tau$, if needed.

representationForm: A string specifying which of the three possible representations ('parameterized', 'tabulated' or 'equation'), is being used for the gating variable. This is discussed in more detail below.

representation: The attributes setting the behaviour of the gating variable, stored in an object derived from the type GatingVariableRepresentation.

The approach here uses a common model organization that maps different styles of representation into one of three types of representations: a parameterized form, a tabulated form, and a catch-all equation form. The three forms are subtypes of GatingVariableRepresentation, and are described below. Each gating variable can be independently described using one of the three main forms. This gives the user freedom to model categories of channels that use combinations of, say, a gating variable described using a parameterized form and another gating variable described using a tabulated form. Regardless of the particular form, every gating variable has two rate functions, one for forward (in Hodgkin–Huxley's framework, the $\alpha(V)$ function) and one for backward (the $\beta(V)$ function). Each of the three subtypes of GatingVariableRepresentation therefore has two attributes, forward and backward.

### (a) *The parameterized form*

The version of the Hodgkin–Huxley model of voltage-gated channels used here is based on a generalized form of the Hodgkin–Huxley equations for rate variables (Hodgkin & Huxley 1952). It is characterized by three main assumptions about the equations for channel conductance and the forward and backward rate functions. First, the model assumes an Ohm's law relationship between current and conductance, with

$$I_k = G_k(E_k - V_m), \tag{A2}$$

$$G_k = \bar{g}_k X^{X_{\text{power}}} Y^{Y_{\text{power}}} Z^{Z_{\text{power}}}, \tag{A3}$$

where $E_k$ is the equilibrium potential for the channel of type $k$, $\bar{g}_k$ is a normalization constant, and $V_m$ is the membrane potential. Second, the gating variables are calculated from differential equations of the form

$$\frac{\mathrm{d}X}{\mathrm{d}t} = \alpha(V_m)(1 - X) - \beta(V_m)X, \tag{A4}$$

$$\frac{\mathrm{d}Y}{\mathrm{d}t} = \alpha(V_m)(1 - Y) - \beta(V_m)Y, \tag{A5}$$

$$\frac{\mathrm{d}Z}{\mathrm{d}t} = \alpha(V_m)(1 - Z) - \beta(V_m)Z. \tag{A6}$$

The third and final assumption is that the voltage dependence of the $\alpha(V_m)$ and $\beta(V_m)$ functions are each expressed using a differential equation that can ultimately be rewritten in the form

$$\alpha(V_m) = \frac{A + BV_m}{C + H\exp\left(\dfrac{V_m + D}{F}\right)}. \tag{A7}$$

The attribute category in the ParameterisedForm data structure determines which of several variations of this equation should be presented to the user. We have so far determined the following categories: 'General Parameterised Form', 'Hodgkin–Huxley exponential', 'Hodgkin–Huxley sigmoid', 'Hodgkin–Huxley linoid', 'Sigmoidal $m_\infty$ and constant $\tau$', and 'Borg–Graham'.

Equation (A7) given above is the most general; it is called the 'General Parameterised Form'. This same formula for $\alpha(V_m)$ and $\beta(V_m)$ can represent the three common forms of the Hodgkin–Huxley (1952) equations:

*Exponential*: By setting $B = 0$, $C = 0$, $D = -V_0$, $F = -B$ and $H = 1$, we obtain the Hodgkin–Huxley exponential form,

$$\alpha(V_m) = A \exp\left(\frac{V_m - V_0}{B}\right). \tag{A8}$$

*Sigmoid*: By setting $B = 0$, $C = 1$, $D = -V_0$, $F = B$ and $H = 1$, we obtain the Hodgkin–Huxley sigmoid form,

$$\alpha(V_m) = \frac{A}{\exp\left(\frac{V_m - V_0}{B}\right) + 1}. \tag{A9}$$

*Linoid*: By setting $A = -AV_0$, $B = A$, $C = -1$, $D = -V_0$, $F = B$ and $H = 1$, we obtain the Hodgkin–Huxley linoid form,

$$\alpha(V_m) = \frac{A\,(V_m - V_0)}{\exp\left(\frac{V_m - V_0}{B}\right) - 1}. \tag{A10}$$

Other forms, including some besides those developed by Hodgkin & Huxley (1952), can also be represented using this general approach.

We assume that the user interface for a tool such as Modeler's Workspace will provide a fill-in-the-blanks facility that triggers on the user's choice from a pull-down list for the category attribute, and display one of the specialized equation forms for user input. This will help present the user with a slightly more familiar format, for those cases when they are interested in using one of the common types of equations.

ParameterisedForm does not have attributes related to tabular fill or interpolation. The reason is that ParameterisedForm is intended to describe gating variables using a particular, general equation. The issue of how the generalized equation is implemented in a simulator (for example, whether it is internally turned into a table of numbers, as in the *tabchannel* of GENESIS') is a simulator-specific issue. Some simulators may not turn the generalized equation into a table at all. Therefore, it is not suitable to provide attributes related to table translation in the ParameterisedForm data structure. However, it *is* appropriate for the TabulatedForm, and therefore, TabulatedForm does have attributes that let a user specify how a table should be expanded by interpolation; see § (b) below.

### (b) *The tabulated form*

The TabulatedForm data structure is used to represent gate variables with tabulated data. This can be useful in cases where a modeller has experimental data characterizing the behaviour of a channel gate. The representation here provides for separate tables for the forward and backward rate functions. Each can be a one- or two-dimensional table of floating-point quantities stored in elem, together with variables startingValue and endingValue that express the range of the independent quantity in the table. Both startingValue and endingValue are lists of one-to-two floating-point numbers, to handle the case of 1D and 2D tables.

The interpolation attribute, using the InterpolationParameters structure, allows a user to express expectations about how a given table should be expanded by interpolation. This is useful if the user has a small number of data points, but expects those data points to be used to fill a larger table using a particular interpolation method. The possible values of method are restricted to a small number of predefined methods, to help ensure that users see the same results in different simulation programs.

The TabulatedForm data structure still assumes equations (A2)–(A6) apply. However, the tabular format allows the expression of gating variables that have forms other than that of equation (A7), if so desired.

### (c) *The equation form*

The EquationForm data structure stores rate functions expressed as formulas. It has the following parameters for this purpose: equation, a text string representing a formula; startingVoltage and endingVoltage, floating-point values representing the range of voltage values for which the formula is valid; assign, a list of ParameterAssignment data structures for assigning values to symbolic parameters used in the formula; and an optional divisions attribute that can be used to specify the number of divisions into which the equation should be discretized over the range startingVoltage to endingVoltage. The syntax of the expression language permitted in the equation attribute is similar to that used in the C and Java languages, with some minor differences.

As is true for the tabulated form, the EquationForm data structure still assumes equations (A2)–(A6) apply. However, the equation format allows the expression of gating variables that have forms other than that of equation (A7).

Readers may wonder why formulas are expressed as text strings and not using MathML (W3C 2000). The reason is that using MathML would require a complicated parser in the tools using the representations. Formulas expressed as text strings are much closer to the forms already being used by simulation tools such as GENESIS and NEURON, making it that much simpler to implement translators that can take the representation and convert it into a form suitable for these simulation environments.

Finally, another point is worth clarifying in this context. EquationForm is not intended to express an equation that is used to fill a TabulatedForm table. The three representational forms (ParameterisedForm, TabulatedForm, EquationForm) are independent methods for describing channel behaviour. A GENESIS or NEURON simulator plug-in translating an EquationForm representation may generate a

```
<neuroml version = "1">
  <hhvoltagegatedchannel name = "Traub et al. 1991 sodium channel"
      channelType = "sodium"    currentType = "sodium"
      cellRestingMembranePotential = "0.06"
      cellRestingMembranePotential_units = "V"
      temperature = "32"    temperature_units = "C"
      Q10Factor = "1"    scalingFactor = "1"
      equilibriumPotential = "0.055"    equilibriumPotential_units = "V">
    <version timeStamp = "2000–10–30 18:40 PST"  versionNumber = "1.0"/>
    <description>
      A description of the Traub et al. 1991 sodium channel </description>
    <notes> The neuron model here refers to the channel used within the
          CA3 pyramidal cell model. Parameter values are drawn from
          the GENESIS version of the Traub et al. model (traub91chan.g)
          as developed by David Beeman circa 1999. </notes>
    <listOfAuthors>
      <author xlink:href = "14849220795111111@dbase.publisher.com"/>
      <author xlink:href = "14849220795111122@dbase.society.org"/>
    </listOfAuthors>
    <listOfReferences>
      <reference xlink:href = "150292208045213D@dbase.univ.edu"/>
    </listOfReferences>
    <listOfIons>
      <ion name = "Na">
        <listOfGatingVariables>
          <gatingVariable name = "X" exponent = "2" dependency = "voltage"
              instantCalculation = "no" format = "alpha/beta"
              representationForm = "Parameterised">
            <representation>
              <forward category = "General Parameterised Form"
                A = "35000" B = "0" C = "0" D = "0.005" F = "–0.010" H = "1"/>
              <backward category = "General Parameterised Form"
                A = "7000" B = "0" C = "0" D = "0.065" F = 0.020" H = "1"/>
            </representation>
          </gatingVariable>
          <gatingVariable name = "Y" exponent = "1" dependency = "voltage"
              instantCalculation = "no"    format = "alpha/beta"
              representationForm = "Parameterised">
            <representation>
              <forward category = "General Parameterised Form"
                A = "128" B = "0" C = "0.0" D = "0.080" F = "0.018" H = "1"/>
              <backward category = "General Parameterised Form"
                A = "4000" B = "0" C = "1.0" D = "–0.003" F = "–0.005" H = "1"/>
            </representation>
          </gatingVariable>
        </listOfGatingVariables>
      </ion>
    </listOfIons>
  </hhvoltagegatedchannel>
<neuroml>
```

Figure A2. An XML data stream encoding a channel model.

script-language function that embodies the EquationForm equation, or it may create a discretized, tabulated representation of the funtion. The latter case would be likely for GENESIS. However, if the translation program does produce a tabular representation of the equation, it should not then store the results in a TabulatedForm representation in the model definition. The different forms store the *user's* input; they should not be modified by a simulator plug-in translation program.

### (d) *XML example*

Traub *et al*. (1991) developed a simplified model of hippocampal pyramidal neurons based on physiological data. Their model used a variety of ionic conductances (i.e. ion channels in NeuroML terms; see the discussion in §6(d)). We present here as an example, a NeuroML-based description of their model of a sodium channel.

The Traub *et al*. (1991) model of the sodium channel has the following characteristics:

(i) The activation variable rate functions are given by

$$\alpha(V) = \frac{0.32(13.1 - V)}{\exp\left(\frac{13.1 - V}{4}\right) - 1}, \tag{A11}$$

$$\beta(V) = \frac{0.28(V - 40.1)}{\exp\left(\frac{V - 40.1}{5} - 1\right)}. \tag{A12}$$

(ii) The inactivation variable rate functions are given by

$$\alpha(V) = 0.128 \exp\left(\frac{17 - V}{18}\right), \tag{A13}$$

$$\beta(V) = \frac{4}{1 + \exp\left(\frac{40 - V}{5}\right)}. \tag{A14}$$

(iii) We assume that most of the experimental data used by Traub *et al*. (1991) were either gathered at a temperature of 32 °C, or else that Traub *et al*. (1991) adjusted the quantities in their model to match this temperature. We also assume temperature scaling factors of unity.

(iv) We assume an equilibrium potential for sodium of 0.055 V.

(v) We express the resting cell membrane potential as being measured relative to an extracellular potential of zero, and having a value of −0.06 V. This follows the convention used in simulation environments such as GENESIS; see the discussion of the Traub *et al*. (1991) model by Bower & Beeman (1998, Chapter 7).

(vi) We have converted the original units used in the paper to SI units; this, combined with the change in convention for the cell resting membrane potential, affects the values for the variables $A$, $B$, $C$, $D$, $F$ and $H$ in the parametrized representation used below.

Figure A2 shows the main portion of an example XML data stream encoding this channel model using NeuroML. The outermost container in the XML encoding in figure A2 is the tag neuroml, identifying the contents as using Neural Open Mark-up Language. The attribute version indicates that the content is formatted according to version 1 of the definition of NeuroML.

The next-inner container is a single hhvoltagegated-channel element that serves as the highest-level object in the model. The model has a name, 'Traub *et al*. 1991 sodium channel'. The various general attributes of a voltage-gated channel model in NeuroML are expressed as attributes of the hhvoltagegatedchannel element. A <version> element provides information about the version of the model.

The rest of the model representation consists of gating variable descriptions for the sodium ion. There are two gating variables, corresponding to the activation and inactivation rate functions shown in equations (A11)–(A14). The parameters $A$, $B$, $C$, $D$, $F$ and $H$ are the terms in equation (A7).

## REFERENCES

Beeman, D., Bower, J. M., De Schutter, E., Efthimiadis, E. N., Goddard, N. & Leigh, J. 1997 The GENESIS simulator-based

neuronal database. In *Progress in Neuroinformatics* (ed. S. Koslow & M. Huerta). Hillsdale, NJ: Lawrence Earlbaum Associates.

Biron, P. V. & Malhotra, A. 2000 XML Schema Part 2: Datatypes (W3C Working Draft 7 April 2000). See http://www.w3.org/TR/xmlschema-2/.

Bosak, J. & Bray, T. 1999 XML and the second-generation web. *Scient. Am.* **280**, 89–93.

Bower, J. M. & Beeman, D. 1998 The book of GENESIS: exploring realistic neural models with the GEneral NEural SImulation System, 2nd edn. New York: Springer Verlag.

Bray, T., Paoli, J., Sperberg-McQueen, C. M. & Maler, E. 2000 Extensible Markup Language (XML) 1.0, W3C Recommendation 6-October-2000. See http://www.w3.org/TR/REC-xml.

Burns, G. A. P. C. 2001 Knowledge management of the neuroscientific literature: the data model and underlying strategy of the NeuroScholar system. *Phil. Trans. R. Soc. Lond.* B (This issue.)

Cannon, R. 2002 'The CATACOMB simulation environment', *Neurocomputing.* (In the press.)

DeRose, S., Maler, E. & Orchard, D. 2000 XML Linking Language (XLink) Version 1.0, W3C Proposed Recommendation 20-December-2000. See http://www.w3.org/TR/xlink.

Eriksson, H.-E. & Penker, M. 1998 *UML toolkit*. New York: John Wiley.

Fallside, D. C. 2000 XML Schema Part 0: Primer. See http://web4.w3.org/TR/xmlschema-0/.

Forss, J., Beeman, D., Eichler-West, R. & Bower, J. M. 1999 The Modeler's Workspace: a distributed digital library for neuroscience. *Future Generation Comput. Syst.* **16**, 111–121.

Fujimoto, R. M. 1990 Parallel discrete event simulation. *Commun. ACM* **33**, 31–53.

Gardner, D., Abato, M., DeBellis, R., Erde, S. M., Knuth, K. H. & White, T. 1999 Common data model 2000: open methods for neuroscience data description and interchange. *Soc. Neurosci. Abstr.*, p. 1910.

Gardner, D., Knuth, K. H., Abato, M., Erde, S. M., White, T., DeBellis, R. & Gardner, E. P. 2001*a* Common data model for neuroscience data and data model exchange. *J. Am. Med. Informatics Assoc.* **8**, 17–33.

Gardner, D., Abato M., Knuth K. H., DeBellis R. & Erde S. M. 2001*b* Dynamic publication model for neurophysiology databases. *Phil. Trans. R. Soc. Lond.* B (This issue.)

Goddard, N. H. 1994 Rochester connectionist simulation environment. In *Neural network simulation environments* (ed. J. Skrzypek), Chapter 10, pp. 187–207. Kluwer Academic.

Goddard, N. H. & Hood, G. 1997 Parallel GENESIS. In *Computational neuroscience: trends in research 1997* (ed. J. M. Bower), pp. 911–917. New York: Plenum.

Goddard, N., Hood, G., Howell, F., Hines, M. & De Schutter, E. 2001 NEOSIM: portable large-scale plug and play neuronal modelling. *Neurocomputing* **38–40**(1–4), 1657–1661.

Hines, M. 1984 Efficient computation of branched nerve equations. *Int. J. Biomed. Computing* **15**, 69–76.

Hines, M. & Carnevale, N. T. 1997 The NEURON simulation environment. *Neural Comput.* **9**, 1179–1209.

Hodgkin, A. L. & Huxley, A. F. 1952 A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol.* **117**, 500–544.

Howell, F. W., Dyhrfjeld-Johnsen, J., Maex, R., Goddard, N. H. & De Schutter, E. 2000 A large-scale network model of the cerebellar cortex using PGENESIS. *Neurocomputing* **32**, 1041–1046.

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. & Kitano, K. 2001*a* The ERATO systems biology workbench: an integrated environment for multiscale and multitheoretic simulations in systems biology. In *Foundations of systems biology* (ed. K. Kitano). MIT Press.

Hucka, M., Shankar, K., Beeman, D. & Bower, J. M. 2001*b* The Modeler's Workspace: making model-based studies of the nervous system more accessible. In *Computational neuroanatomy: principles and methods* (ed. G. Ascoli). Humana Press.

Lamport, L. 1994 *LaTeX: a document preparation system*. Menlo Park, Calif.: Addison-Wesley.

Mirksy, J. S., Nadkarni, P. M., Healy, M. D., Miller, P. L. & Shepherd, G. M. 1998 Database tools for integrating and searching membrane property data correlated with neuronal morphology. *J. Neurosci. Meth.* **82**, 105–121.

Nelson, M. & Rinzel, J. 1998 The Hodgkin–Huxley model. In *The Book of GENESIS*, 2nd edn (ed. J. M. Bower & D. Beeman), pp. 29–49. Springer-Verlag.

Oestereich, B. 1999 *Developing software with UML: object-oriented analysis and design in practice*. Harlow, UK: Addison-Wesley.

Peterson, B. E., Healy, M. D., Nadkarni, P. M., Miller, P. L. & Shepherd, G. M. 1996 ModelDB: an environment for running and storing computational models and their results applied to neuroscience. *J. Am. Med. Informatics Assoc.* **3**, 389–398.

Rall, W. & Agmon-Snir, H. 1998 Cable theory for dendritic neurons. In *Methods in neuronal modeling*, 2nd edn (ed. C. Koch & I. Segev), Chapter 2, pp. 27–92. MIT Press.

Segev, I. & Burke, R. E. 1998 Compartmental models of complex neurons. In *Methods in Neuronal Modeling*, 2nd edn (ed. C. Koch & I. Segev), Chapter 3, pp. 93–136. MIT Press.

Shepherd, G. M., Mirsky, J. S., Healy, M. D., Singer, M. S., Skoufos, E., Hines, M. L., Nadkarni, P. M. & Miller, P. L. 1998 The Human Brain Project: neuroinformatics tools for integrating, searching and modelling multidisciplinary neuroscience data. *Trends Neurosci.* **21**, 460–468.

Thompson, H. S., Beech, D., Maloney, M. & Mendelsohn, N. 2000 XML Schema Part 1: Structures (W3C Working Draft 7 April 2000). See http://www.w3.org/TR/xmlschema-1/.

Traub, R., Wong, R. K. S., Miles, R. & Michelson, H. 1991 A model of a CA3 hippocampal pyramidal neuron incorporating voltage-clamp data on intrinsic conductances. *J. Neurophysiol.* **66**, 635–650.

Vibert, J.-F., Alvarez, F. & Kosmidis, E. K. 2001 XNBC V9: A user friendly simulation and analysis tool for neurobiologists. *Neurocomputing* **38–40**(1–4), 1715–1723.

W3C 2000 W3C's Math Home Page. See http://www.w3.org/Math/.