CBI Simulator Framework: Multiscale modelling with multiple solvers

Cornelis, H.[1,*], Coop, A. D.[2], Rodriguez, A. L.[3], Beeman, D.[4], Bower, J. M.[5].

1. Cornelis H. Department of Neurophysiology, Catholic University of Leuven, Leuven, 3000, Belgium
2. Coop A. D. Director of Flight Simulation, Merindah Energy.
3. Rodriguez A. L. Research Imaging Institute, University of Texas Health Science Center at San Antonio, San Antonio, TX, United States
4. Beeman D. Department of Electrical, Computer, and Energy Engineering, University of Colorado, Boulder, CO 80309
5. Bower J. M. Barshop Institute for Longevity and Aging Studies, 15355 Lambda Drive, University of Texas Health Science Center, San Antonio, Texas 78245

* E-mail: Corresponding Author Hugo.Cornelis@gmail.com

Abstract

Extension of simulator functionality may result in the unintentional creation of a monolithic software platform. Analysis suggests that the difficulties of simulator extensibility faced by many software developers originates in a primary focus on the biological and mathematical implementation of a model, at the expense of considering the underlying software architecture. This greatly increases the difficulty of extending simulator software code to support new functionality required by efficient multi-scale modeling. It limits software development efforts, thus simulator functionality, and ultimately simulator extensibility.

Here, axioms are proposed that define the domain of computational neuroscience. They have been extracted from over twenty years of simulator and model development by the global GENESIS developers community and provide a logical framework that organizes the approach to multi-scale simulation that is described here. This framework, referred to as the CBI federated software architecture, underpins the reconfigured GENESIS simulator. This has resolved many problems associated with multi-scale modeling that previously existed with a simulator that incrementally became an unmanageable monolith.

The approach to development of simulator structure and function is outlined by describing essential components of the CBI federated software architecture for multi-scale modeling.

Careful consideration of the issues identified greatly facilitates the development of a simulator capable of transparently supporting multi-scale biological models across levels ranging from the ionic and molecular to complete systems.

1 Introduction

Scientific understanding of problems involving the interaction between different spatial and temporal scales is very challenging. Over the last 30 years several schemas have been introduced to address perceived requirements of spatio-temporal computer-based neurophysiological modelling and simulation. One of the earliest was that of Marr [15]. He identified three independent ways to understand visual processing, through: (a) computational theory–what does the system do or what problems does it solve? (b) representation and algorithm–how does the system solve the problem? and (c) hardware implementation–how is the system physically realized? More recently, Churchland and Sejnowski [5] have also proposed three levels: (1) level of analysis, includes Marr's three levels, (2) level of processing, for example the levels of processing in the visual pathway, and (3) level of structural organization, for which they proposed seven (ascending) levels that extend from molecules to synapses, neurons, local networks, layers and columns, topographic maps, and systems.

start with an introduction of (1) multiscale modeling and (2) a user-workflow. Maybe a user-workflow should simply be explained first.

Recently, we proposed a five step "ideal user workflow", or user workflow. This workflow organizes the sequence of activities typically employed for the development and simulation of a computational model, including data generation and analysis. It consists of five steps: construct model, design experiment, run simulation, analyze output, and iterate [12]. Such a workflow allows the object under investigation, the tools used to perform the investigation, and the operations performed during a simulation to be distinguished. These categories correspond to the three approaches to computational modelling identified by Marr: (1) computational theory, (2) representation and algorithm, and (3) level of structural organization. For clarification, in the biological domain we refer to 'multi-level' or level, in the modelling domain to 'multi-scale' or scale, and in the software domain to 'multi-layer' or layer.

This is out of place. Which categories?

Here, we extend our workflow paradigm by introducing the organizing principle of a "cognitive workflow". This workflow describes the process whereby an abstraction is transformed into an implementation by the conversion of a mental model into the physical implementation of an actual simulation. One cognitive workflow that facilitates such a path is given by the top row in Figure 1 (Biology $\rightarrow$ Computer Simulation). In this figure, the column to the left gives the spatial scale and lists levels of biological organization (object under investigation), the central column gives the scale of the mathematical implementation or algorithm (operations performed during the investigation), and the column to the right lists the equivalent layers of software and hardware (tools used to perform the investigation). The rows in each column give examples of possible simulator functionality at the given level.

Each step in the cognitive workflow can be expanded into a matrix where columns span the range of structural detail available for simulation. As an example, Figure 2 expands the Biology step of the cognitive workflow to give a matrix describing a taxonomy of possible models. Each row illustrates a

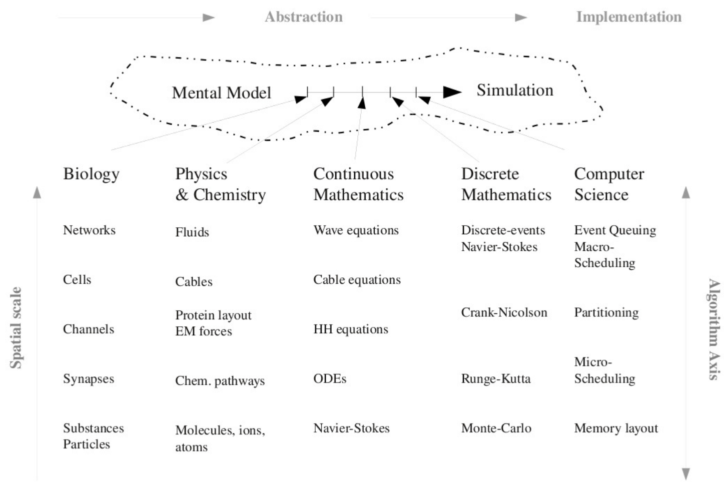


Figure 1: **A Multi-Scale Path from Mental Model to Simulation.**

set of models that spans the range of model detail available at the given level of structural detail. The other steps in the cognitive workflow can similarly be expanded to collectively give a comprehensive framework for multi-scale modelling.

The multidimensional framework implied by Figures 1 and 2 is based on the relationships between the schemas outlined by Marr, Churchland, and Sejnowski and the user and cognitive workflows described here. We use these relationships to develop a novel and simplifying approach to the simulation of multi-scale models.

If it is accepted that: (1) a scale specific approximation can be abstracted at each level of a system and disjointly partitioned from its environment

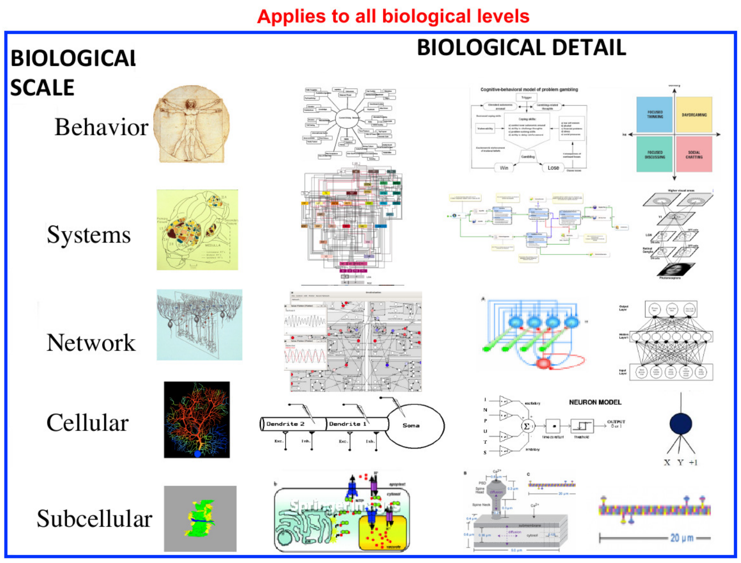


Figure 2: **A Multi-Scale Model Taxonomy.**

[23, 14], and (2) the biological levels listed in Figure 1 are plausible, then it is apparent that a realistic neuron model comprised of at least a soma and dendrites, that includes channels and synapses, is a multi-level model simulated at multiple scales.

We describe the features of the reconfigured GENESIS simulator (G-3–http://genesis-sim.org) relevant for multi-scale simulation. Together with several scripting examples, we show how primary components of G-3 transparently support multi-scale simulation at the biological levels of networks, cells, channels, and synapses (see Fig. 1).

2 Methods

2.1 Structural Overview of the CBI Architecture

The Computational Biology Initiative Federated Software Architecture (CBI architecture) is defined as a modular paradigm that places stand-alone software components into a set of logical relationships [12]. In doing so, it defines a modular framework that provides the necessary parts of a simulator. The architecture takes its name from the Computational Biology Initiative at the University of Texas at San Antonio, where development was first initiated.

A schema identified by separation of concerns (see [12] for details) is expanded in Figure 3 to give the modules that form the building blocks of the CBI architecture. The schema retains the four quadrants of simulator functionality identified by the separation of concerns, including separation between control and data and the notions of high-level representations for biology and low-level data for numerics (Fig. 3: indicated by vertical and horizontal dashed lines, respectively).

We note that the integrity of the simulator is maintained by only allowing interactions between vertically and horizontally located modules. Diagonal interactions are forbidden as they foster the mixing of functionality across different levels and typically lead to monolithic software applications. Ultimately, it is this partitioning of simulator functionality that lies at the core of simulator extensibility and provides the architectural foundation for transparent multi-scale simulation.

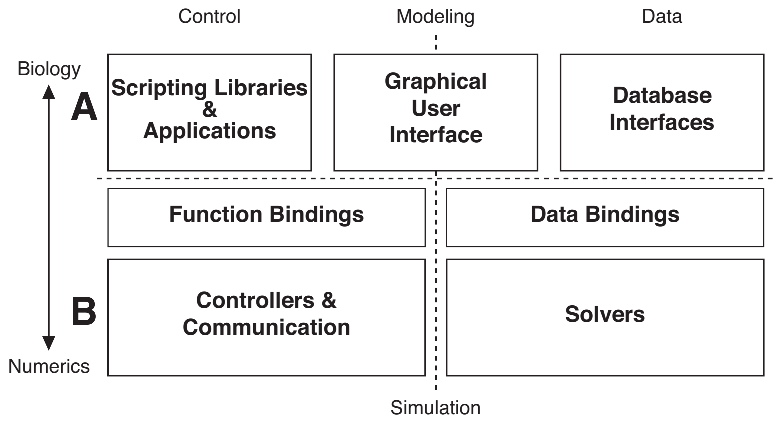


Figure 3: **Overview of a federated software architecture:** The primary functional modules defined for the CBI federated software architecture are illustrated. Control modules are given to the left and data modules to the right. A. The top layer contains conceptual data and controls representations of the biology of a model. B. The bottom layer contains representations that are numeric and thus close to the hardware. The middle or intermediate layer bridges between these 'biological' and 'numerical' layers in a CBI compliant simulator. Importantly, Control (Scripting Libraries & Applications) and Data (Database Interfaces) modules can interact either directly or via the Graphical User Interface.

We refer to the CBI architecture as being 'federated' as it extends the modular approach associated with the development of single applications to the functional integration of otherwise independent applications. Federation aims to provide a unified interface to diverse applications and ideally make them look like a single system to the user. In doing so, it provides transparency, heterogeneity, a high degree of function, autonomy for the underlying federated sources, extensibility, openness, and the possibility of highly optimized performance (see [10]). Here, extensibility is defined as a system design principle where an implementation takes into consideration future developments. An extensible system is one that includes mechanisms for expanding or enhancing the system with new capabilities without having to make major changes to system infrastructure.

Enhanced application interoperability and performance is available through the use of flexible high-level scripting languages that support diverse workflows, low-level application programmer interfaces (API)s, and application binary interfaces (ABI)s (see [13]).

In summary, the CBI architecture provides a template for software development that, at its core, contains a simulator. Additionally, the modularity and layering of the architecture simplifies connection to independent applications related to model construction and instantiation and the analysis and display of simulation output.

2.1.1 Behavioural View of the CBI Architecture

Figure 4 expands Figure 3 to give more detail of the structural relationships between the different modules and sub-modules of the CBI architecture (previously described in [12]). The behavior of the CBI architecture is defined by the functional and dynamic connectivity provided by these individual modules. This behavior is now introduced within the context of the user workflow.

During the phase of model construction prior to simulation, users combine models from files and databases. When the simulation runs, the numerical

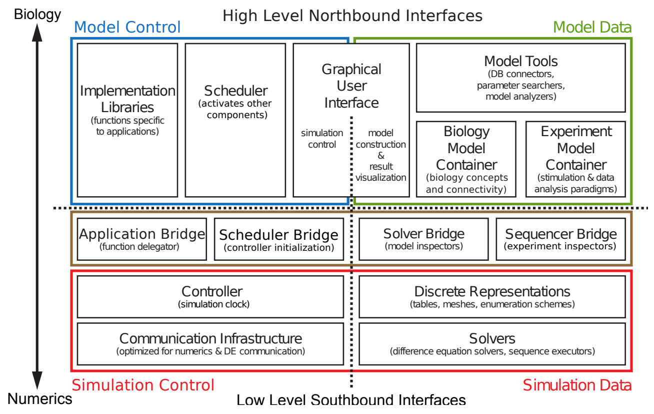

Biology
High Level Northbound Interfaces
Model Control
Model Data
Implementation Libraries
(functions specific to applications)
Scheduler
(activates other components)
Graphical User Interface
simulation control
model construction & result visualization
Model Tools
(DB connectors, parameter searchers, model analyzers)
Biology Model Container
(biology concepts and connectivity)
Experiment Model Container
(stimulation & data analysis paradigms)
Application Bridge
(function delegator)
Scheduler Bridge
(controller initialization)
Solver Bridge
(model inspectors)
Sequencer Bridge
(experiment inspectors)
Controller
(simulation clock)
Discrete Representations
(tables, meshes, enumeration schemes)
Communication Infrastructure
(optimized for numerics & DE communication)
Solvers
(difference equation solvers, sequence executors)
Simulation Control
Simulation Data
Numerics
Low Level Southbound Interfaces

Figure 4: **Detailed view of the Computational Biology Initiative federated software architecture.** Relationships of sub-modules within each of the primary functional modules given in Figure 3 are illustrated. North bound interfaces group and conceptualize the details of the modules and interact with south bound interfaces of higher level modules. Steps 1–3 of the user workflow induce data cycling between the upper layers (blue and green boxes) and lower layers (red box). Ultimately, the two layers interact to implement a single simulation. By design, any type of model including multi-scale models will exhibit this data cycle. In contrast to previous versions of GENESIS and other neuronal simulation platforms, the design of G-3 is fundamentally modular, separating different functional components of the simulator into their own modules. Importantly, G-3 separates model descriptions from their mathematical representations. This simplifies the implementation of new **Solvers** and other run-time software components. The communication infrastructure connects different **Solvers** to simulate different parts of a model and can 'upscale' or 'downscale' numerical data as needed. Additional detail is given in [12].

Solvers perform the calculations of a simulation, and save the output back to files and databases. When these two steps are combined they imply a cyclic data flow from files and databases to the **Solvers**.

In more detail, as a result of Steps 1–3 of the user workflow (construct model, design experiment, run simulation), data flows both through and between software components. The cycle described here is for a single neuron simulation, however, all scales of model will exhibit similar activity.

Once Step 3 is completed, data generated by dynamic states of the model are available through databases and files. Consequently, there is no requirement to query software components that deal with numerical data. This provides an operational barrier to assist with preventing incremental conversion of a CBI compliant simulator to its monolithic equivalent.

In Step 4 of the user workflow (output), while maintaining the integrity of the separation of concerns, the availability of any model data from the **Biology Model Container** and the functionality of **Scripting Libraries & Applications** can be used to connect the CBI architecture to external tools, for example Matlab (to implement output analysis).

At Step 5 of the user workflow (iterate), simulation output data and model parameters and structure available in the model containers can be combined, for example, to provide automated script generators for the creation of batch simulations. Importantly, at this stage of the workflow, the back-ends of the CBI architecture (indicated below the dotted horizontal line in Figures 3 and 4) are unavailable.

2.2 GENESIS 3.0

The software architecture developed for G-3 supports a capacity for federated and modular software development that directly enables multi-scale modelling. Furthermore, its structure also supports the direct interaction with, or embedding of, models at different levels of scale. During model construction this is accomplished through the reconfiguration of the internal model storage format, while during simulation run-time it occurs through specific software modules that create intermediary representations of variables present at multiple scales (described in more detail below).

Several explicit features of G-3 allow new software components to be rapidly incorporated into the simulator: (1) modularity of the CBI architecture supports the development of new **Solvers** independently of other software components. This encourages full focus on the mathematical aspects of **Solvers** and their implementation and leads to better optimization, (2) the *DeveloperPackage* facilitates integration of new **Solvers** into the build system of G-3 enabling immediate regression testing, (3) easy extension of the configuration of the internal model storage format to establish new declarative model tokens and parameters recognized by new **Solvers**, (4) easy construction of an interface between the internal data storage format and the core of new **Solvers**, and finally, (5) a communication software component that creates simulation run-time interfaces between **Solvers**.

Solvers currently available in G-3, starting at the highest level, include: **DES**, a discrete event queuing and distribution infrastructure for network modelling, **Heccer**, a single neuron solver, **Chemesis-3**, a simple reaction-diffusion system solver, and **Experiment**, which provides simulation-time components for current injections, time-tables, and output.

2.2.1 The Model Containers

Biology Model Container Provides a solver-independent internal storage format for models. Stores biological entities and end-user concepts and annotates the structures and concepts of a model with biological, physical, and mathematical quantities. By 'containing' the biological model, the **Biology Model Container** relieves the implementation of the numerical core of a solver from the pressure of user-friendly model representation.

Because all model components are interactive parts of a single model-container [12], existing and new G-3 **Solvers** can be incorporated to simulate different aspects of the same model. This *scale-linking* functionality of G-3 instantiates dedicated run-time software components that contain intermediary representations of solved variables able to pass both *upscale* and *downscale* values to **Solvers** operating at different levels of scale. It is through this computational mechanism that the **Biology Model Container** supports transparent construction and simulation of multi-scale models.

Experiment Model Container Stores a model of a stimulus paradigm and desired output parameters. It also defines and stores a hierarchical sequence of stimulus-related actions (e.g. start and stop time of a current injection) and their dependencies.

Finally, we note that the model containers define models in a purely declarative, extensible language.

2.2.2 The Neurospaces Description Format

The Neurospaces Description Format (NDF) file format developed for the model containers was originally designed as a file-based user-editable declarative interface to the *hsolve* method used by GENESIS-2 [7][2]. Subsequently, its single neuron modelling functionality was expanded with network modelling capabilities.

The NDF file format is indicated by the `.ndf` file name extension. This format integrates and extends the previous GENESIS `.p` and `.g` declarative file formats. NDF files are purely declarative and can be stored in a model database. In an NDF file, a model exists as a hierarchy of tokens, each representing a biological component. Each component is associated with a set of physical processes and can have parameters and shared variables (described in more detail below). This feature allows the NDF format to support both multi-level and multi-scale modeling.

G-3 supports the construction of NDF files. These files define the different components of a model. They simplify and expedite model development by being shared between and imported into other NDF files, or used to create NDF libraries. NDF libraries can be linked and used to store model lineages, thereby curating and representing different levels of neuronal morphology and/or electrophysiological function. These relationships support and control the modular construction of multi-scale models as any given NDF file can depend on other NDF files.

Overview of an NDF File An NDF file contains four sections. They are not necessarily filled, but must be present in the given order. Files have the following general form (described in more detail below).

```
#!/usr/local/bin/neurospacesparse
//-*- NEUROSPACES -*-
// default location for file comments
NEUROSPACES NDF
IMPORT
    FILE <namespace> "<directorypath>/<filename.ndf>"
     . . . <other files may be imported as required>
END IMPORT
PRIVATE_MODELS
    ALIAS <namespace>::/<source label> <target label> END ALIAS
        . . . <other aliases may be defined as required>
END PRIVATE_MODELS
PUBLIC_MODELS
    CELL <morphology name>
        SEGMENT_GROUP segments
            . . . <morphological details>
        END SEGMENT_GROUP
    END CELL
END PUBLIC_MODELS
```

The header section typically starts with a UNIX *interpreter* sequence. It gives the system specific absolute path to the **Model Container** stand-alone executable. This is followed by optional comments. The text `NEUROSPACES NDF` marks the start of model definitions.

The `Import` section uses the `FILE` keyword to declare dependencies on other NDF files.

The `Private Models` section defines models that are private to a single NDF file. They may depend on imported public models of other files.

The `Public Models` section defines models that are visible to the NDF files into which they are imported. Public models may depend on private models or be hard coded.

Physical Processes and Model Parameters in NDF Mathematical models can be used to describe physical processes underlying the functional activity of biological entities by associating the physical processes with parameters (fixed during a simulation) and variables (computed during a simulation).

As an example, consider a calcium pool Ca^{2+} whose concentration follows an exponential decay according to the equation:

$$\frac{\mathrm{d}[\mathrm{Ca}^{2+}]}{\mathrm{d}t} = B \cdot I_{\mathrm{Ca}^{2+}} - \frac{[\mathrm{Ca}^{2+}]}{\tau} \qquad (1)$$

where $I_{\mathrm{Ca}^{2+}}$ is an ionic current flowing into a segment and B is a decay constant (sometimes referred to as β), often found by fitting experimental data [4].

In an NDF file calcium decay can be represented as follows:

```
POOL Ca_concen
  PARAMETERS
    PARAMETER ( concen_init = 75.5e-6 ),
    PARAMETER ( TAU = 0.01 ),
    PARAMETER ( VAL = 2.0 ),
    PARAMETER ( BETA = FIXED
                (PARAMETER ( value = 1.98e+11 ),
                 PARAMETER ( scale = 1.0)  ))
  END PARAMETERS
END POOL
```

In the example above, each parameter has a value that is fixed during the model construction phase. In the general case a parameter can either be a constant value, point to a parameter of another component, or have the value of a function. For example:

```
PARAMETERS
   PARAMETER ( DIAMETER = 0.3 ),
   PARAMETER ( LENGTH = ..->LENGTH ),
   PARAMETER ( SEED = SERIAL () )
END PARAMETERS
```

where `DIAMETER` has a constant value, `LENGTH` is inherited from the parent component (the '..' notation is a reference to the parent component), and `SEED` has a value defined by the function `SERIAL()`. Importantly, neither the NDF file format nor the model container make a syntactical distinction between the parameters and variables of a physical process. For example, `LENGTH` is neither defined as a parameter with a fixed value nor as a solved variable. This logic is employed as a numerical value may be a parameter in one model but a variable in another.

When one biological entity modulates the behavior of another, their physical processes share one or more variables. For example, in the case of a 'pool' containing calcium ions, a transmembrane calcium current may raise the concentration of the pool. In this example, the current (`I`) is a variable common to both the conductance and the pool. Such variables, when shared between multiple physical processes, are declared using the `BINDABLES` keyword.

```
BINDABLES
  INPUT I
END BINDABLES
```

Assuming a calcium current is present under the label `../cat`, it can be bound to a pool by using a `BINDINGS` clause:

```
BINDINGS
  INPUT ../cat->I
END BINDINGS
```

Multi-scale Models in NDF The NDF file format is loosely based on GENESIS-2 objects [4], and supports an extensible set of tokens, including `POOL`, `CHANNEL`, `SEGMENT`, `CELL`, `PROJECTION`, `POPULATION`, and `NETWORK`, amongst others, that allow the construction of models ranging from subcellular dynamics up to the network level. Other tokens are simple containers for the parameters of procedural algorithms that run inside the **Biology Model Container** during the model construction phase. For instance these tokens can define a three dimensional layout of physical processes (`Grid3D`) or bind variables of selected physical processes based on their spatial parameterization (`VolumeConnect`).

2.2.3 A Single-Neuron Solver: Heccer

Heccer is the G-3 reimplementation of the compartmental solver (*hsolve*) of the GENESIS-2 simulator [6]. It employs the Crank-Nicolson solution to integrate the cable equation of a neuronal morphology and the Hodgkin-Huxley equations of their transmembrane currents [20, 1, 6]. **Heccer** also integrates exponentially decaying calcium concentrations and Nernst potentials but more elaborate calcium models must be interfaced with other **Solvers**.

2.2.4 Chemesis-3

Biochemical pathways in neuronal modeling are complex networks of interacting ion concentration pools. The G-3 implementation for simulation of biochemical pathways currently under development is called **Chemesis-3** and has a dedicated optimized implementation to represent networks parameterized with biochemical pathways.

Chemesis-3 was recently adapted from a GENESIS-2 ‘add-on’ library of objects for modelling biochemical reactions, second messengers, and calcium dynamics [3].

The separation of the mathematical functions of the GENESIS-2 objects lead to the implementation of **Chemesis-3** as a stand-alone **Solver** for G-3. The functions currently implemented in **Chemesis-3** include: *rxnpool*–a concentration pool that interacts with reactions and diffuses to other pools, *conservepool*–a mass conservation based pool, it computes the difference between the total of all molecules (a model parameter, rest state) and diffused molecules, divided by compartment volume, *reaction*–provides standard forward/backward chemical reactions between pools of molecules, and *diffusion*–computes the flux in molecules between two pools.

The second step in the integration of **Chemesis-3** provides a set of appropriate definitions in the **Model Containers** for **Chemesis-3** to recognize supported NDF tokens. These definitions are given in three small text files: *kinetics.yml*: defines groups of pools of molecules and their chemical interactions, *reaction.yml*: defines reactions between pools, and *species.yml*: supports the definition of parameters of a chemical species.

2.2.5 Schedulers and User Interfaces

SSPy and SSP Both the model containers and **Solvers** are stand-alone G-3 components. To be useful, they must be 'glued' together and activated correctly, such that they can coordinate during a single simulation. This is exactly what the Simple Scheduler in Python (**SSPy**) and the Simple Scheduler in Perl (**SSP**) do. **SSPy** and **SSP** exploit the sophistication of the Python and Perl scripting languages to load software components on demand and activate them from a configuration file. They achieve this by loading dynamically those software components referenced in their configuration. Because the scheduler does not have any computational load, its implementation is very simple and highly configurable. Other configurations can connect **SSPy** or **SSP** to different modelling services or **Solvers**.

In summary, **SSPy** and **SSP** are highly configurable components that glue other software together and synchronize their activity during a simulation.

G-Shell The G-3 shell environment (**G-Shell**) integrates G-3 components and makes their functions available through an interactive environment available from a command line. It is started from a system shell with the *genesis-g3* command. After the **G-Shell** completes its startup procedure, a command prompt (`genesis` >) appears in the terminal window. This indicates the existence of an interactive G-3 session. To provide backward compatibility, the **G-Shell** also interfaces with the GENESIS-2 Script Language Interpreter

(SLI).

The Neurospaces Studio The Neurospaces Studio (**Studio**) is a GUI front-end to the **Biology Model Container**. It supports the browsing and visualization of a model and the concepts of its connectivity[18]. Note that the **Studio** is not a graphical editor or construction kit. External applications such as **neuroConstruct** should be employed for these purposes [11].

2.3 Interactive Query and Simulation

Coded in Perl, the **G-Shell** provides an integrated interactive environment for the model containers, **Heccer**, **SSP**, **DES** and the **Studio**.

Integration of the **G-Shell** with the **Biology Model Container** enables real-time analysis of the quantitative and structural aspects of a neuronal morphology. The library of model components installed with the **Biology Model Container** provides, for example, the definition of a model Purkinje cell in the file *cells/purkinje/edsjb1994.ndf*. The command:

```
ndf_load cells/purkinje/edsjb1994.ndf
```

makes this model Purkinje cell available for interactive analysis. A similar command exists for backward-compatible loading of GENESIS-2 SLI scripts with *sli_load*. A *pynn_load* to interface to the PyNN network modeling environment [9] is currently under development.

If a dendritic segment contains a synaptic channel, it can be stimulated with a precomputed spike train stored in a file named, for example, *event_data/events.yml*:

```
set_runtime_parameter /Purkinje/segments/b1s06[182]/Purkinje_spine_0/head/par/s
   EVENT_FILENAME ``event_data/events.yml''
```

Following the addition of an output for the somatic membrane potential:

```
add_output /Purkinje/segments/soma Vm
```

a simulation can conveniently be started using:

```
run /Purkinje 0.1
```

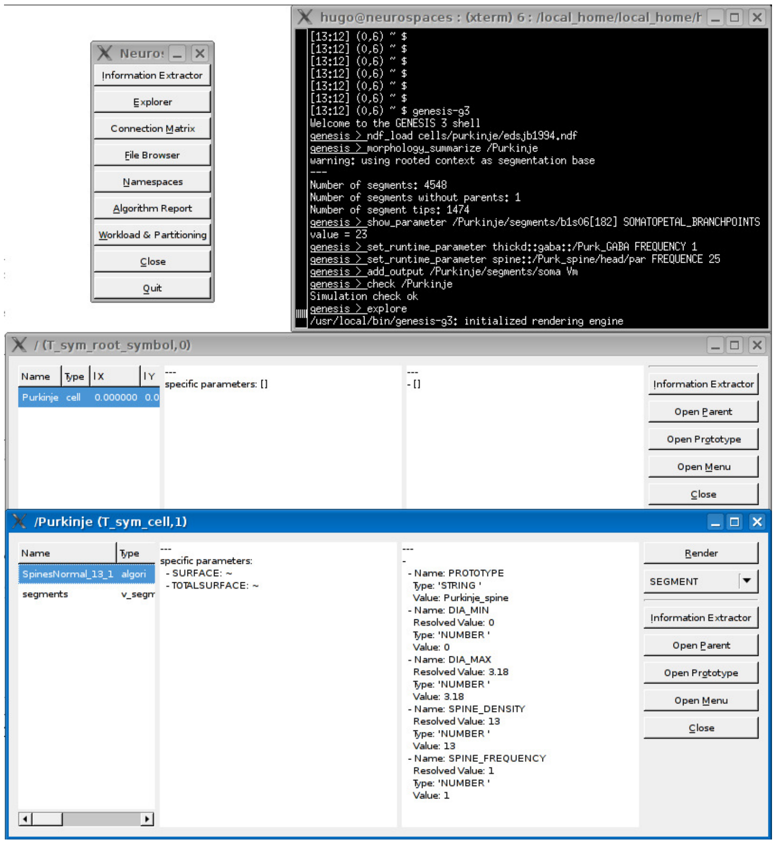
This simulation produces the somatic response to given dendritic stimulation in a file named, by default, */tmp/output*.

To query the parameters of the stimulated compartment the model can be analyzed using the graphical front-end of the **Studio** with the command *explore*.

Figure 5 shows sample output of running this command. Other capabilities of the **Studio** include rendering morphologies in three dimensions and generating overviews of network models (not shown).

2.4 Simple Scripting

Python is a high-level scripting language that uses modules to group related functions. G-3 employs Python's module system to group the functions that provide an interface to each of the simulator's software components. As an



hugo@neurospaces : (xterm) 6 : /local_home/local_home/h

```
[13:12] (0,6) ~ $
[13:12] (0,6) ~ $
[13:12] (0,6) ~ $
[13:12] (0,6) ~ $
[13:12] (0,6) ~ $
[13:12] (0,6) ~ $
[13:12] (0,6) ~ $ genesis-g3
Welcome to the GENESIS 3 shell
genesis > ndf_load cells/purkinje/edsjb1994.ndf
genesis > morphology_summarize /Purkinje
warning: using rooted context as segmentation base
---
Number of segments: 4548
Number of segments without parents: 1
Number of segment tips: 1474
genesis > show_parameter /Purkinje/segments/b1s06[182] SOMATOPETAL_BRANCHPOINTS
value = 23
genesis > set_runtime_parameter thickd::gaba::/Purk_GABA FREQUENCY 1
genesis > set_runtime_parameter spine::/Purk_spine/head/par FREQUENCE 25
genesis > add_output /Purkinje/segments/soma Vm
genesis > check /Purkinje
Simulation check ok
genesis > explore
/usr/local/bin/genesis-g3: initialized rendering engine
```

/ (T_sym_root_symbol,0)

Name	Type	IX	IY
Purkinje	cell	0.000000	0.0

```
---
specific parameters: []
```

```
---
- []
```

/Purkinje (T_sym_cell,1)

Name	Type
SpinesNormal_13_1	algori
segments	v_segm

```
---
specific parameters:
  - SURFACE: ~
  - TOTALSURFACE: ~
```

```
---
-
 - Name: PROTOTYPE
   Type: 'STRING '
   Value: Purkinje_spine
 - Name: DIA_MIN
   Resolved Value: 0
   Type: 'NUMBER '
   Value: 0
 - Name: DIA_MAX
   Resolved Value: 3.18
   Type: 'NUMBER '
   Value: 3.18
 - Name: SPINE_DENSITY
   Resolved Value: 13
   Type: 'NUMBER '
   Value: 13
 - Name: SPINE_FREQUENCY
   Resolved Value: 1
   Type: 'NUMBER '
   Value: 1
```

Figure 5: **Querying a model in GENESIS 3.0.** The **Studio** is a G-3 simulator component that can be used to query the parameters of individual compartments in a multi-compartment model neuron. The **Studio** also renders 3D morphology of dendrites and generates overviews of network models.

example, the G-3 Python module *nmc* contains functions to simplify the storage of neuron models in computer memory. The module is a simple front-end to the **Biology Model Container**. Likewise, *Heccer* is a wrapper module for the **Heccer** component. We note that Python bindings for **DES** to facilitate network modelling are in development.

2.5 Scripting Chemesis-3

The following example, *cal1.ndf*, shows how to implement a G-2 Chemesis tutorial in G-3. It creates a single compartment that contains interaction between calcium and a buffer.

```
NEUROSPACES NDF
PUBLIC_MODELS
  KINETICS cal1
    PARAMETERS
      PARAMETER ( DIA = 24e-4 ),
      PARAMETER ( LENGTH = 24e-4 ),
    END PARAMETERS
    POOL somaCa
      BINDABLES OUTPUT concen END BINDABLES
      PARAMETERS
        PARAMETER ( concen_init = 0.001 ),
        PARAMETER ( "UNITS" = 1e-3 ),
      END PARAMETERS
    END POOL
    POOL somaCabuf
      BINDABLES OUTPUT concen END BINDABLES
      PARAMETERS
        PARAMETER ( concen_init = 0.003 ),
        PARAMETER ( "UNITS" = 1e-3 ),
      END PARAMETERS
    END POOL
    POOL somabuf
      BINDABLES OUTPUT concen END BINDABLES
      BINDINGS
        INPUT ../somaCabuf->concen,
      END BINDINGS
      PARAMETERS
        PARAMETER ( concen_init = 0.153 ),
        PARAMETER ( concen_total = 0.153 ),
      END PARAMETERS
    END POOL
    REACTION somacabufrxn
      BINDABLES INPUT concen, OUTPUT concen END BINDABLES
      BINDINGS
        INPUT ("substrate") ../somaCa->concen,
        INPUT ("substrate") ../somabuf->concen,
        INPUT ("product") ../somaCabuf->concen,
      END BINDINGS
      PARAMETERS
        PARAMETER ( FORWARD_RATE = 1e2 ),
        PARAMETER ( BACKWARD_RATE = 0.5 ),
      END PARAMETERS
    END REACTION
  END KINETICS
END PUBLIC_MODELS
```

The length and diameter of the pools required to complete the model take on their implicit default values of:

```
  PARAMETER ( DIA =  ..->DIA ),
  PARAMETER ( LENGTH =  ..->LENGTH ),
```

The following commands illustrate how the **G-Shell** and the **Studio** may be employed to load *cal1.ndf* into the **Biology Model Container** for exploration of this model:

```
  ndf_load chemesis/cal1.ndf
  explore
```

3 Results

3.1 Addressing Scheme Algorithm

The different solvers contributing to a multi-scale simulation can hold equivalent variables. These variables exist because two or more physical processes of the model have been bound using the BINDINGS in an NDF file. During a simulation, *the addressing scheme algorithm* enables the **Solvers** and the

Communication Infrastructure to relate any equivalent variables these components contain. As this happens during a simulation, *the addressing scheme algorithm* must be highly optimized to maximize run-time performance.

The information required to identify a variable (and hence its address) is obtained from the model stored in the **Biology Model Container**. To achieve this, the **Biology Model Container** defines an addressing scheme that is able to refer to or ‘address’ the variables of every model component. These addresses provide the ‘hooks’ for attaching “experimental protocols” (e.g. stimulus and output specifications) to the **Solvers**. The addressing scheme is not necessarily dependent on the NDF format. It is more general in the sense that it provides an interface to external software applications such as GUIs and databases.

In G-3, the algorithm for taking the simulation-time address of a computed variable consists of three steps.

Step 1 Translates NDF names to a hierarchical namespace. The scheme defines a symbolic syntax for the hierarchical addressing of model components and their parameters and variables. The syntax allows a user to express the address of a variable, for example, the address of the Ca^{2+} concentration in a segment, with the name ‘soma’:

```
/soma/Ca->conc
```

Step 2 The second step is the translation of symbolic references to IDs and is implemented in the **Biology Model Container**. There, the hierarchical namespace (created during step 1) is translated to simulation wide unique tuples of the form [integer, type]. The integer identifies the physical process and the type identifies the variable containing the concentration level.

Step 3 The third step of the addressing algorithm translates [integer, type] tuples to tuples of the form [solver instance, internal ID]. This step is low-level and implemented in the **Solvers**.

After this step of the algorithm, an interface has been defined that enables the communication port of the solver that attaches to the given variables to be obtained. This interface allows a solver instance internal ID to be obtained for directly addressing the variable.

The addressing scheme algorithm allows **Solvers** to communicate through the tuples [solver instance, internal ID]. The advantage is that the **Biology Model Container** does not have to be in memory during simulation run-time. This removes any requirement for expensive computation on symbolic or hierarchical addresses during simulation run-time.

Software may choose to implement the interface between solvers using memory pointers for maximum efficiency, or, alternatively, using a low-level function that reads and writes the variable. This feature may be useful in a parallel implementation for example.

3.2 DES: Action Potential Propagation Abstraction

The Discrete Event System (**DES**) is used for the abstract modelling of action potentials, a functionality required for the running of network simulations. Action potentials generated at the soma of one neuron are translated to a discrete event that is delivered to the post-synaptic targets of a connected neuron. **DES** also associates 'secondary' data with a connection, for example, propagation delay, synaptic weight, or other data specific for the model.

Internally, **DES** contains two subcomponents, one for event distribution that contains a connectivity matrix, the second for event queuing. In the CBI architecture, the **DES** event queuer is a separate solver that simulates action potential propagation in an efficient way. The **DES** event distributor provides communication between the solvers during the simulation.

The separation of the discrete event functionality from the rest of the simulator facilitates customization for the modeling of sophisticated learning rules, especially those related to STDP, diffusion, and spillover [19, 22].

3.3 Multi-Scale Network Modeling

Network modelling was previously possible in G-2 [6]. To simulate networks of cells the user must manually initialize the numerical solver of G-2 for every cell in the network. Careful G-2 script coding was required to prevent memory exhaustion errors. The complexity of the G-2 procedural scripts contrasts with the simplicity of the configuration of the G-3 addressing scheme given below.

The **SSP** software component defines several software regression tests with network models. One of the simplest network models used in this test has a ‘source’ neuron that delivers a spike to two ‘target’ neurons during the simulation .

To simulate this simple network, the simulation configuration has to associate each neuron model in the **Model Container** with **Heccer**. This association will then create a **Heccer** instance before the simulation starts. As with G-2, in G-3 this is currently a manual operation. However, the integration of the user workflow into G-3 defines the scope of user actions and in the case of network modeling, allows the **Heccer** instances, once created, to share internal data structures as required, thereby transparently reducing overall memory consumption.

In G-3 the action potential propagation through the fibers of a projection is simulated by the event queuing component of **DES**. The **G-Shell** syntax is:

```
ndf_load tests/networks/spiker4.ndf
solverset "/network/target1 => heccer"
solverset "/network/target2 => heccer"
solverset "/network/source => heccer"
solverset "/network/projection => des"
```

3.4 Multi-Scale Scripting

In a typical multi-scale model different numerical solution methods are required to solve the different types of mathematical equations associated with the different scales of the model. In G-3 the cable equation and ion currents are numerically solved with implicit Crank-Nicolson integration using **Heccer**. Calcium models are numerically solved with **Chemesis-3**. The association between a given model and its **Solvers** can be configured by the user. For example to associate the model named "/Purkinje" with the solver **heccer**, the command is:

```
solverset "/Purkinje => heccer"
```

To use **Chemesis-3** to simulate a complex network model of biochemical pathways, here called *cal1*, a user would type in the G-3 shell:

```
solverset "/cal1 => chemesis3"
```

In the case where a network of biochemical pathways is defined inside a single neuron model, a user would have to type two commands with wildcards to associate the correct solver with each component of the model, for example:

```
solverset "/**/cal1 => chemesis3"
solverset "/Purkinje/**[!cal1] => heccer"
```

4 Discusion

Simulation provides a framework to organize our understanding of biological systems. Historically, the development of neuronal simulation software for the construction of morphologically detailed neuron models and small networks has been instigated by research projects that specifically addressed complementary technical and scientific questions [17]. These software systems have been both highly successful and continue to grow in complexity through cycles of research project extension. However, after more than twenty years of extending their functionality, usually by the direct incorporation of source code into the core of the simulator (typically comprising a single solver), code structures have become so complicated that it is increasingly difficult, if not impossible, to easily continue extension of simulator functionality. Ultimately, the resulting stand-alone applications become 'monolithic'. In these conditions it is a considerable challenge for a neuroscientist lacking the necessary mathematical and computational skills to extend a model. In practice, the complexity and monolithic nature of the software results in a non-scalable software architecture [8].

At the core of the problem is the lack of distinction in many contemporary simulators made between biological data, numerical data, and control operations during the software architecture development stage of simulator design. The majority of such simulators remain primarily focused on the biological and mathematical aspects of a model. As they do not respect all three of Marr's levels, i.e. the third of Marr's levels ("hardware" implementation) is not addressed during software design, the complexity of multi-scale simulation is greatly increased.

We note that just as the 'hardware' of a biological system should be accounted for in the cognitive process of model development, so must it be accounted for during the simulation of the model. In a simulator this can be achieved by the software architecture and its organization of simulator functionality. Making the cognitive model concordant with simulator functionality removes many of the problematic aspects of multi-scale modeling.

It has previously been proposed that [21]: (1) because of the the many different structural levels of organization and the fact that models rarely span more than two levels, a more comprehensive understanding of a system requires many different types of model, and (2) intermediate level models must necessarily simplify with respect to the structural properties of lower level elements. We consider this view to be based on the difficulty of employing a monolithic simulator to address a multi-level problem.

To clarify these issues, we turn to a discussion in [14]. Systems theory shows that regardless of the number of scales used to conceptualize a system, it is not possible to know whether the system has been completely conceptualized. This is a consequence of changing scales as when this occurs it adds, deletes, and reconfigures patterns that are the content of the associated conceptualizations. There is no way to know or learn that all patterns have been conceptualized at one or more scales, as completeness is not possible.

Nevertheless, the use of multiple scales of conceptualization provides a way to asymptotically approach completeness.

However, as [14] continues, at any scale of conceptualization, properties are always aggregated at least as simple objects. Thus, a system at any scale of conceptualization is an approximation or loose congruence of the actual system. This is not an 'abstraction' of the system as that means leaving things out of a model with the understanding that what is absent can always be reinserted. What is "left out" of a scale specific conceptualization of a system are patterns (simple objects, relationships, and their aggregations) that might be available at other scales. Such patterns cannot be returned to a scale specific approximation of a system, regardless of resolution or field of view. This means that 'missing' content has no place in a conceptualization at a given scale as it was never available in the first place.

According to principles of systems theoretic analysis (see [23, 14]), it is clear there are numerous phenomenological approaches to the development of multi-scale models suited to the transformation of a mental model into a computational simulation. We resolve this problem by expansion of the cognitive workflow as illustrated in Figure 1.

The cognitive workflow is an adaption of a previously described user workflow that organizes the sequence of necessary steps typically employed by a person in developing a computational model and employing simulation to generate data for subsequent analysis [12]. In this sense it is a depiction of a sequence of operations, declared as the work of a person or a group of persons (Belhajjame et al., 2002). Similarly, the cognitive workflow describes the process whereby an abstraction is transformed into an implementation by the conversion of a mental model into a simulation. This is made possible due to the recent reconfiguration of the GENESIS simulator.

The reconfiguration complied with the CBI architecture which is based on a separation of concerns. It partitions data and control functions from software layering that separates high level biological data from low level mathematical operations. The immediate benefit of this partitioning (see [12] for details), is transparent support of multi-scale simulation.

The approach is based on the implementation of a model for simulation being contained by a single software entity (here a single **Biology Model Container**). This single entity interfaces with the multiple solvers required to address the different scales of a model, but it is only needed during the construction of the model. A communication infrastructure enables communication at runtime between these solvers and thus across different levels of a simulation. In this paradigm, the requirement for multiple models is removed by implementation of multiple solvers. As a consequence, simplification of intermediate models with respect to lower levels as proposed by [21] may be eliminated.

To this end, G-3 supports the construction of multi-scale network and system models based on realistic neuronal components. In support of this function, the G-3 **Biology Model Container** applies specific algorithms to instantiate systems level connectivity models and convert them to mathematical representations. This relieves the mathematical solvers from the algorithmic pressures typical for rich heterogeneous model structures. At the same time, intermediate simulation system modules, already described as the mechanism used to link **Chemesis-3** and linear cable solvers, are used to share values and functions between larger scale network simulations and lower scale models.

From a technical point of view, the incorporation of simulation modules that specifically involve spatial dimensions is, in principle, much more difficult than the integration of biochemical kinetic models. In fact the accurate representation of space and time for multi-scale simulations is an open research area. However, the explicit provision of software modules for the intermediary representation of solved variables allows the comparison and evaluation of different methods of scale-linking, for instance by comparing different resolutions of a fixed grid with adaptive grid intermediary representations [16]. In effect, this allows the use of multi-scale G-3 models as a framework to compare different technical strategies.

The developing paradigm of multi-scale modelling reported here enables the generation of a taxonomy of models in computational neurobiology based on both the biological scale of a model and the level of detail it aims to represent. By accounting for each of the levels proposed in [15] during development of a simulator, it is possible to transparently resolve many of the technical and conceptual issues currently surrounding the successful simulation of multi-scale models.

Acknowledgements

Hugo Cornelis was partially supported by the CREA Financing program (CREA/07/027) of the K.U.Leuven, Belgium, EU. Both Hugo Cornelis, Allan D. Coop were partially supported by NIH grant 5 R01 NS049288-08 to James M. Bower. We thank the Australian and Belgian Governments for their support of employable people.

Armando L. Rodriguez and David Beeman, were supported by NIH grant 5 R01 NS049288-08 and 3 R01 NS049288-06S1 to James M. Bower.

We also thank the Computational Biology Initiative at UTSA (http://www.cbi.utsa.edu) for their support when installing, using, and updating G-3 on their computers.

References

[1] Hodgkin A and Huxley A. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500–544, 1952.

[2] David Beeman. *20 Years of Computational Neuroscience*, chapter A History of Neural Simulation Software. Springer, 2012.

[3] K.T. Blackwell. Evidence for a distinct light-induced calcium-dependent potassium current in hermissenda crassicornis. *Journal of Computational Neuroscience*, 9(2):149–170, September 2000.

[4] James M. Bower and David Beeman, editors. *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System.* Springer-Verlag, New York, second edition, 1998.

[5] PS Churchland and TJ Sejnowski. *The Computational Brain.* Bradford Books: Cambridge MA, 1992.

[6] Hugo Cornelis and Erik De Schutter. Tutorial : Simulations with Genesis using Hsolve. Course material, November 2002. Available from http://genesis-sim.org/filemanager/active?fid=95, accessed 2011 Dec 11.

[7] Hugo Cornelis and Erik De Schutter. Neurospaces: Separating modeling and simulation. *Neurocomputing*, 52–54:227–231, 2003.

[8] Jaeger D. Accurate reconstruction of neuronal morphology. In Erik De Schutter, editor, *Computational Neuroscience, Realistic Modeling for Experimentalists*, chapter 6. CRC Press: New York, 2002.

[9] AP Davison, D Brüderle, J Eppler, J Kremkow, E Muller, D Pecevski, L Perrinet, and P Yger. PyNN: A common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2:11, 2008.

[10] Lin E and Haas L. IBM federated database technology. www-128.ibm.com/developerworks/db2/library/techarticle/0203haas/0203haas.html, March 2002.

[11] P Gleeson, V Steuber, and RA Silver. neuroConstruct: A tool for modeling networks of neurons in 3D space. *Neuron*, 54:219–235, 2007.

[12] Cornelis H, Coop AD, and JM Bower. A federated design for a neurobiological simulation engine: The CBI framework for GENESIS 3.0. *PLoS ONE*, 7(1):e28956, 01 2012.

[13] Cornelis H, Rodriguez AL, Coop AD, and Bower JM. Python as a federation tool for GENESIS 3.0. *PLoS ONE*, In press, 2011.

[14] F Heylighen, P Cilliers, and C Gershenson. Complexity and philosophy. *arXiv:*, page cs/0604072v1, 2006.

[15] D Marr. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information.* W. H. Freeman & Co Ltd, 1982.

[16] Bernabeu MO, Bordas R, Pathmanathan P, Pitt-Francis J, Cooper J, Garny A, Gavaghan DJ, Rodriguez B, Southern JA, and Whiteley JP. CHASTE: Incorporating a novel multi-scale spatial and temporal algorithm into a large-scale open source library. *Philosophical Transactions of the Royal Society A*, 367:1907–1930, 2009.

[17] J W Moore. A personal view of the early development of computational neuroscience in the USA. *Frontiers in Computational Neuroscience*, 4:20, 2010.

[18] Eilen Nordlie and Hans Ekkehard Plesser. Visualizing neuronal network connectivity with connectivity pattern tables. *Frontiers in Neuroinformatics*, 3:39, 2009.

[19] Patrick D. Roberts and C.C. Bell. Spike timing dependent synaptic plasticity: mechanisms and implications. *Biological Cybernetics*, 87:392–403, 2002.

[20] I. Segev, J. Rinzel, and G. H. Shepherd, editors. *The Theoretical Foundation of Dendritic Function : Selected Papers by Wilfrid Rall with Commentaries.* MIT Press, Cambridge, MA, 1995.

[21] TJ Sejnowski, C Koch, and PS Churchland. Computational neuroscience. *Science*, 241:1299–1306, 1988.

[22] Nowotny T, Zhigulin VP, Selverston AI, and Rabinovich MI Abarbanel HD. Enhancement of synchronization in a hybrid neural circuit

by spike-timing dependent plasticity. *Journal of Neuroscience*, 23:9776–9785, 2003.

[23] L von Bertalanffy. *General System Theory*. George Braziller: New York, (Revised Edition) edition, 1973.