CHAPTER 6.2

Brain Models on the Web and the Need for Summary Data

Amanda Bischoff-Grethe,[1] Jacob Spoelstra,[2] and Michael A. Arbib[3]
[1]*Center for Cognitive Neuroscience, Dartmouth College Hanover, New Hampshire*
[2]*HNC Software, Inc., San Diego, California*
[3]*University of Southern California Brain Project, University of Southern California, Los Angeles, California*

Abstract

Computational models of the brain have become an increasingly common way in which to study brain function. These models vary in design from the study of cable properties of a single neuron to a large-scale network of neurons for modeling behavior or overall function. These models may be hard-coded in a programming language, such as C, or may be developed in one of many programming environments meant for model design (e.g., NSL, Neuron, GENESIS, and others). However, the model's main point (or the hypotheses behind its construction) may be difficult to ascertain. A user who is able to read the code might be able to determine whether a connection or property exists, but may not know the reason for that property unless it has been documented in the code or a README file. Alternatively, the user may have to refer to other documentation (such as a journal publication) which often does not contain enough detail at the code level. Also, models of similar functions may be written in different programming environments and rely upon different hypotheses. Used in conjunction with a summary database, the Brain Models on the Web database is one way to bring these models together in a common environment for study by both modelers and experimenters alike.

6.2.1 Storing Brain Models

Brain Models on the Web (BMW) is a development, research, and publications environment for models. The main goal of the BMW database is to act as an online repository for brain models and to facilitate the publishing of links between models and the underlying data. The idea is to allow designers not only to explore the structure and code of a model, but also to see the data underlying each assumption or design decision. A user is then able to search the database (or federated databases of summary and empirical data) for new data that either support or contradict the existing "facts" and from there develop a new model to take into account the new evidence. This new model version can be partly or completely based upon code from the previous model, or it may link together two separate models in a brand new way. Multiple versions can be stored within the database, a useful tool when comparing alternative methods. Alternatively, experimenters can start at the data end (which might be an experimental protocol) and from there find models that either have used the data in their design or replicate/contradict that particular data item. These models might produce testable predictions which could lead the way to more experiments. Finally, BMW serves as a repository for "works in progress" for users creating

and conducting experiments on models. It relies upon a Summary Database (SDB) for the summary data which justifies a given model, as well as the storage of assertions made concerning a model. We will describe the relationship between the SDB and BMW in more detail in Section 6.2.2. BMW thus makes it easy to find models, to store data summaries culled from the literature and to keep track of versions as a model evolves.

A model may be characterized by its *conformancy* with BMW standards as belonging to one of five different classes: work-in-progress (WIP), poster, bronze, silver, and gold. WIP models are kept private to the individual or group working on them. These are models that are in development and are not available for public viewing. Once the model is completed, it may be submitted as a poster. A poster model is stored in BMW and is available for public viewing but has not yet been reviewed. A committee or peer-reviewed journal publication would review the submitted poster's scientific merit and grant it full status using a descriptor—bronze, silver, or gold—to indicate that the poster had been reviewed and satisfied certain criteria. Bronze, silver, and gold by no means indicate the quality of the model but are labels for keeping track of how well the model is documented and linked to other databases within the BMW framework. A bronze model simply involves code stored at an ftp site (or within the database for retrieval) with a simple README document describing the model. Silver implies that the user has put some effort into "BMWfying" (pronounced "beamifying") the model; for example, for NSL models, the modules have been constructed in a form acceptable to database storage, and there is some limited documentation on the model. Gold is a fully BMWfied model. All the code has been referenced appropriately within module tables, and full documentation, including summaries and annotations, exist within the SDB. Any documentation must conform to one or more of the following formats: a plain text file, HTML, or PDF.

6.2.2 The BMW-SDB Relationship

The "An Overall Perspective" section in Chapter 6.1 has already made clear the importance of summary databases in general and noted the three summary databases developed to date by the USC Brain Project: the Summary Database developed as part of BMW (referred to as SDB throughout this chapter); NeuroScholar (see Chapter 6.3); and the NeuroHomology database (see Chapter 6.4). Summary data summarize a more complex body of texts (and/or graphics, tables, graphs, simulation results, etc.). The summary may also be linked to excerpts from the source data which we refer to as *clumps*. Because an increasing number of journals are available online, we expect to take advantage of our annotation technology (Chapter 5.4) to link summary statements to the clumps supporting the summary as found within the online document itself.

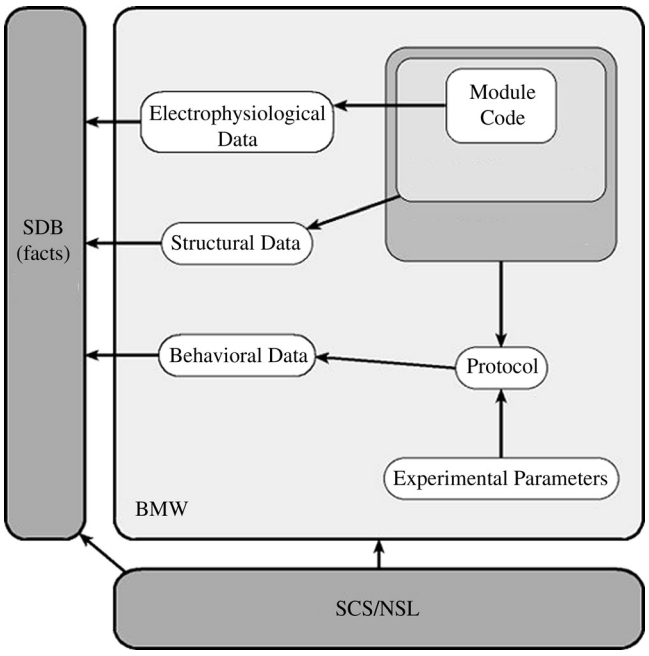


Figure 1 Relationship of the SDB summary database and the remainder of BMW. A model is composed of modules and is used via a model interface which captures aspects of an empirical protocol. These may be supported or tested by links to data stored in the SDB. SCS/NSL is the Schematic Capture System (SCS) developed for NSL. SCS can be used to view data and NSL models and modules schematically in terms of the connections and the structure that comprise them.

Fig. 1 shows the relation of the SDB (the specific summary database developed as part of the BMW effort, focused on data linked to modeling efforts) to the remainder of BMW. The box labeled "BMW" shows that a model is composed of modules and is used via a model interface which captures aspects of an empirical protocol. (See Chapters 1.1 and 2.1 for more on this methodology.) Thus, the design of a module and the way in which modules are combined to form a model may be supported or tested by links to structural, electrophysiological, and other data stored in the SDB, while simulation experiments may yield predictions that can either be posted in the SDB or tested by comparison with summary data in the SDB. At present, BMW-SDB is a relatively small database, and in this chapter we will emphasize the explicit links entered into the database by the modeler. As the database grows, we expect much modeling work to involve searches for data that support or undermine hypotheses used in the model and for empirical results that can test or guide simulation experiments to be conducted with the model. The bottom box refers to the Schematic Capture System (SCS) we have developed for NSL (see Chapter 2.2 for details), which can be used to view data and models schematically in terms of diagrams expressing the way in which models

and modules are formed as connections of the modules that comprise them. We describe the proposed relationship between BMW and SCS in the "Links to SCS" section below.

The data within the SDB function as a method for documenting the model code; they provide justifications for choices made by the modeler when designing the model. This encourages the modeler to be exacting in determining parameters. A parameter value may be chosen either because the value may be documented by the literature summarized in the SDB, or because the value was found by "parameter identification," whether analytical or using multiple simulation runs, which determine what parameter values allow the model to exhibit behavior or other empirical constraints documented in the SDB. To date, most of our work with the SDB is based on previously published documents. This is limiting to some extent, as researchers do not publish all their data, nor do they necessarily provide precise documentation as to the methodology used in generating the data. It is our hope that as the SDB and Repositories of Experimental Data (REDs) are more widely used, researchers will use REDs to store their data and protocols, and the SDB will hold summaries of the RED data.

6.2.3 Reviewing a Model: The Dart Model of Prism Adaptation

Given the broad-brush overview of BMW-SDB provided by Fig. 1, we will now use a specific example of reviewing a model to introduce a number of features embodied in the current system. With the understanding that this provides, we will then present the tables that constitute the actual database design. These include tables for *persons, models, modules, documents, clumps,* and *experiments*. We then present in detail, with exemplary screen dumps, instructions on how to access the database, including material on logging into the database, browsing the database, searching for models and data, and inserting models and data.

Suppose that we have arrived (by means described in later sections) at the page showing Model Versions for the Dart model (Fig. 2). Let us assume we are interested in looking at version 1.1.1 more closely. Clicking on its hyperlink we now begin to reach the guts of the model (Fig. 3). This particular version of the model has been given a gold conformancy status. This means that the model has been constructed to conform with BMW standards; it relies upon the database for storage of not only the model itself, but of its justifications as well. For this particular version, we can explore several options: we can choose to work with the executable model; we can review a tutorial to understand the model; we can view the data associated with the model; we can view only the top-level module; or we can choose to look at all modules which are contained within the model.

By clicking on the Toplevel Module hyperlink, we can explore the hierarchy of this module. Fig. 4 displays the module implementation for the top level of the Dart model, namely, DART_top. Notice that the module itself has a version. This version number may be different from the version number of the model. This may occur for various reasons: the model is new but relies upon previously written modules; the module itself has been updated; or other modules have been updated, representing a different version of the model. For a given module, we are given a list of the associated files; these typically are the code that handles that module's behavior. We can also see a list of models that use this module. In this case, versions 1.1.1 and 2.1.1 of the Dart model currently use the Toplevel module.

Some models will allow users to execute them from the database. As an example, we show the results of a trial run of the Dart model during basic adaptation (Fig. 5; see color insert). "The Dart" model hypothesizes a role for the intermediate cerebellum during prism adaptation. In experiments by Martin *et al.* (1996a,b), both normal subjects and cerebellar patients were required to throw at a target after donning 30° wedge prism glasses. The prisms caused the subjects to initially miss the target by an angle proportional to the prism deflection angle. With subsequent throws, however, normal subjects adjusted until they were once again throwing on target. In contrast, cerebellar patients did not adapt and continued to miss the target. After removing the glasses, the prism gaze-throw calibration remained and subjects made corresponding errors in the opposite direction.

Fig. 5 (see color insert) shows the result of a simulation experiment using a normal subject. Throws are shown in chronological order on the x-axis, while the y-axis indicates horizontal error (the distance by which the target was missed). The simulation allows for two throw strategies (overhand and underhand) and two conditions (prisms-on and prisms-off). In this particular experiment, the model starts with 10 warm-up throws (5 overhand and 5 underhand) before donning the prism glasses. The first throw while prisms are worn (red squares) misses the target proportional to the prism angle, but the model improves with subsequent throws until it once again hits the target. When the prisms are removed (blue squares), however, we observe an after-effect proportional to the adjustment angle, and the model has to re-adjust its normal throwing, in agreement with the results reported by Martin *et al.* (1996b). The graphical interface allows the user to choose a different experimental protocol (for instance, to observe the effect of the adaptation on underhand throwing) and also to adjust both the experimental and model parameters.

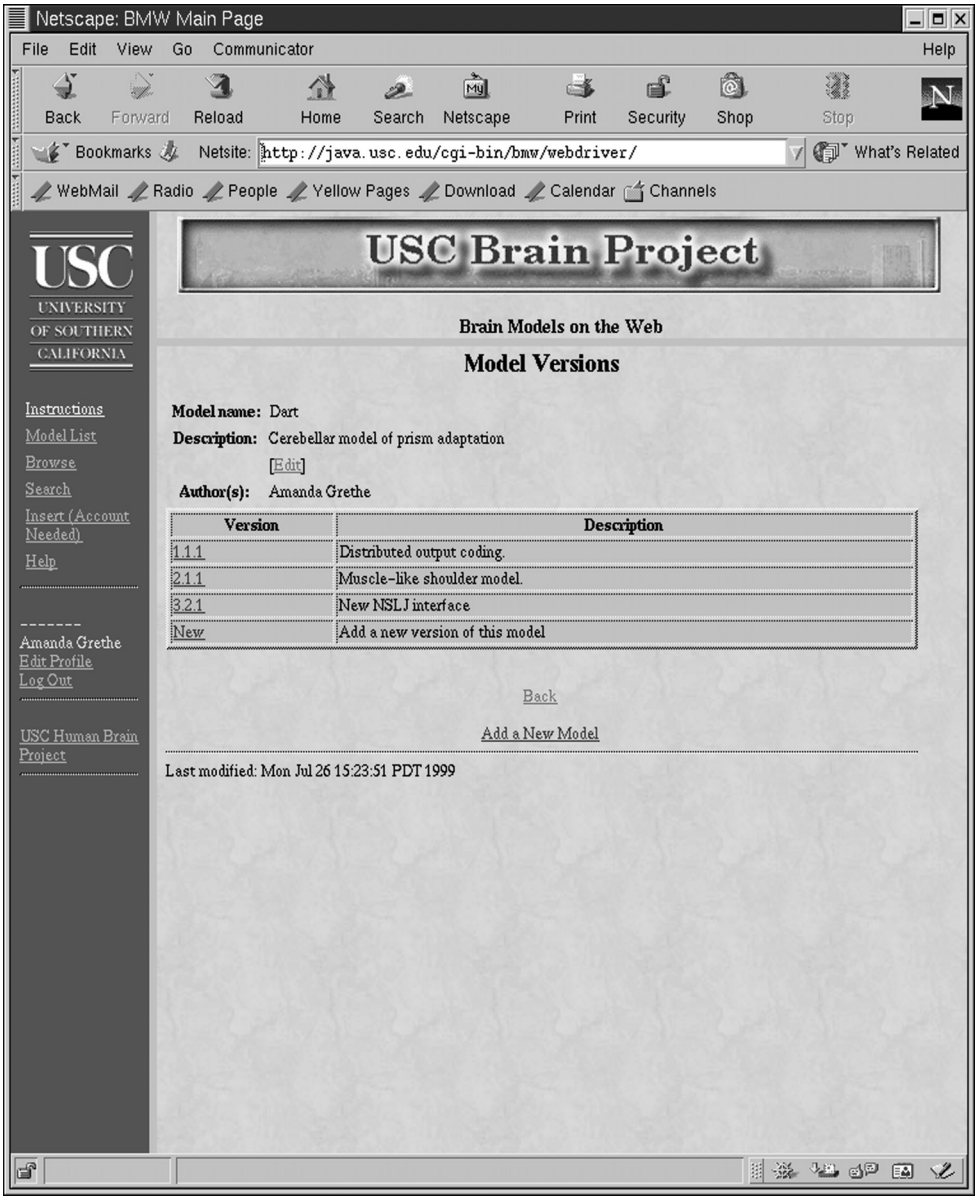
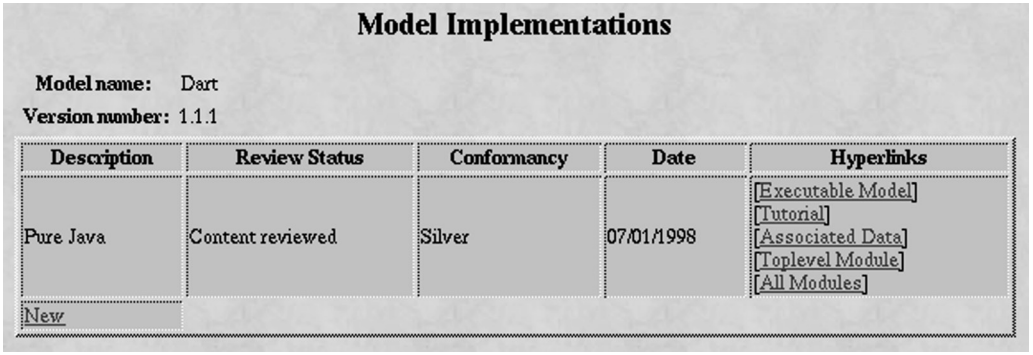


Figure 2 An example of viewing the Brain Models on the Web database of the USC Brain Project. Here, the Dart model has been selected, and information regarding the model versions are presented on the screen. The Dart model has three separate versions. Versioning may be based upon the addition of new material, such as the addition of a shoulder model as seen in version 2.1.1. Clicking the hyperlink related to the model version displays a new page with a list of available information associated with that particular model version.

6.2.4 Database Design

The database was implemented in SQL and is hosted on a Sun workstation running the Informix DBMS. This was done to enhance its ability to connect with other aspects of the NeuroInformatics Workbench and to allow for a consistency across databases contained within the Workbench. There are five main groups of data that may be stored within BMW/SDB:

1. *Persons:* The Person group is fairly self-explanatory; this group contains tables that define persons involved with the database. These include users of the database, modelers, authors of articles, and reviewers.
2. *Models:* The Model group describes tables that relate to models. A model contains descriptive information regarding it, but it also links together the smaller aspects, or modules, which describe it.
3. *Modules:* The Module group contains tables that define modules. These are the building blocks which, when linked together, comprise a model.
4. *Documents and clumps:* Documents are the articles used to support or refute the research. Clumps contain pieces of information that can be used to support (or refute) a model and/or module. These clumps can

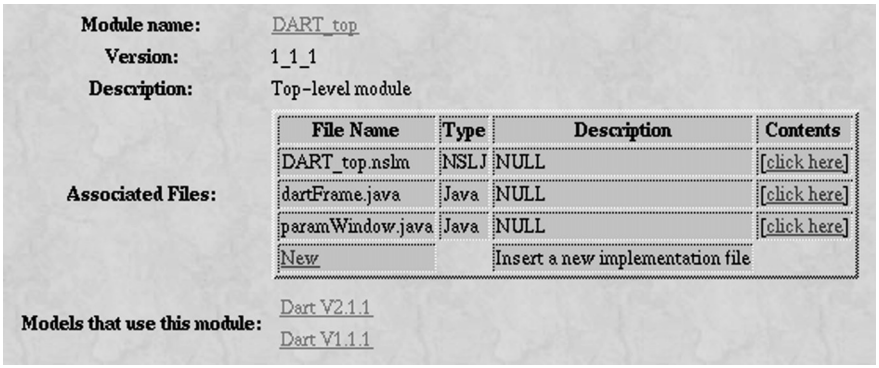


Model Implementations

Model name: Dart
Version number: 1.1.1

Description	Review Status	Conformancy	Date	Hyperlinks
Pure Java	Content reviewed	Silver	07/01/1998	[Executable Model] [Tutorial] [Associated Data] [Toplevel Module] [All Modules]
New				

Figure 3 This panel from a BMW screen lists information regarding a model version along with links to allow the user to run the model, examine a tutorial, review the associated data, or view the modules that were used to create the model.



Module name: DART_top
Version: 1_1_1
Description: Top-level module

Associated Files:

File Name	Type	Description	Contents
DART_top.nslm	NSLJ	NULL	[click here]
dartFrame.java	Java	NULL	[click here]
paramWindow.java	Java	NULL	[click here]
New		Insert a new implementation file	

Models that use this module: Dart V2.1.1, Dart V1.1.1

Figure 4 An example of a module and its associated files. The module was implemented in NSL 3.0 (Java implementation) and hence has several files written in Java and in NSL that are available for viewing. Models using this module are listed at the bottom of the screen.

contain data gleaned from journal articles or experiments or can be hypotheses that the model author puts forth within the model. (As noted earlier, this material constitutes the current SDB embedded within BMW).

5. *Experiments:* The Experiment group contains tables that describe the experiments the model is capable of performing. For example, the Dominey and Arbib (1992) saccade model is able to perform several different kinds of saccade experiments, including simple saccades, memory saccades, and double saccades. The experiments are defined through protocols. Note that a single protocol interface may be used to carry out many experiments.

In the following sections, we will briefly describe these tables and their attributes used to represent these data and provide a graphical representation of their relationships. To better understand the relationships between the tables, we have defined different graphic entities to represent different table types (Fig. 6). We briefly recapitulate the description from the "Overview of Table Descriptions" section of Chapter 3.2.

BMW uses the following table definitions as part of its methodology:

1. *Core table:* A core table allows the storage of information related to a specific concept or entity that is part of the core framework. For example, a person is a basic entity which is a core table describing persons stored within the database.

2. *Set backing table:* A set backing table is used to define the mappings from multiple entries to multiple entries (m-n). As an example, a model may contain many modules. A particular module may be used within many models. A set backing table would therefore be used to describe the mapping between models and modules.

3. *Dictionary table:* These are core tables that describe basic necessary tables used for looking up information. For example, we have previously described levels of conformancy. These would be placed within a dictionary table for reference by other tables within BMW. Dictionary tables are also a way to encourage users to use common nomenclature, but new terms can be added as needed (e.g., platinum as a model conformancy level).

4. *Extendible parent:* An extendible parent table allows individual users to tailor the database to their own needs. For example, a person working with saccadic

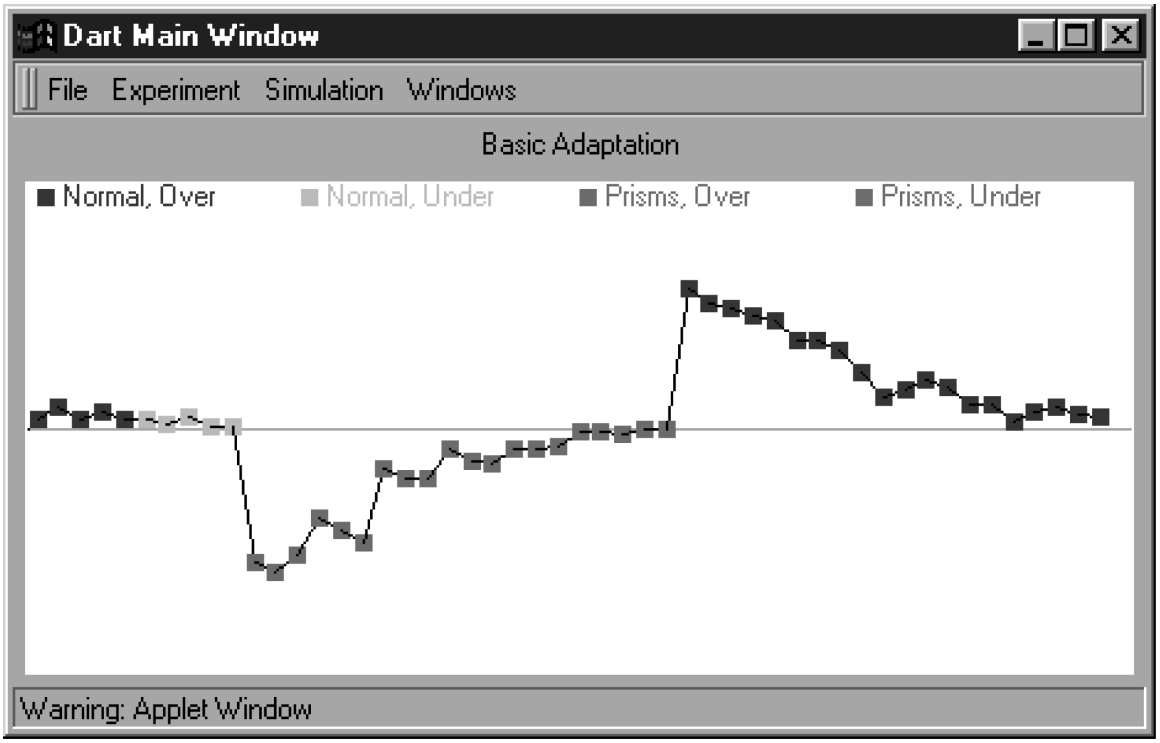


Figure 5 Example of output from the Dart model. The graph demonstrates the effects of wedge prisms on underhand and overhand throwing, along with the adaptation of the user's ability to hit the target when a dart is thrown. In this simulation, the model initially throws both overhand and underhand under normal conditions and is fairly accurate. When prisms are donned, the model must adapt to the related change between sight and movement. Once the model has reached an appropriate level of accuracy, the prisms are removed. The model makes errors in the opposite direction before re-adapting to normal conditions. (see color plates.)

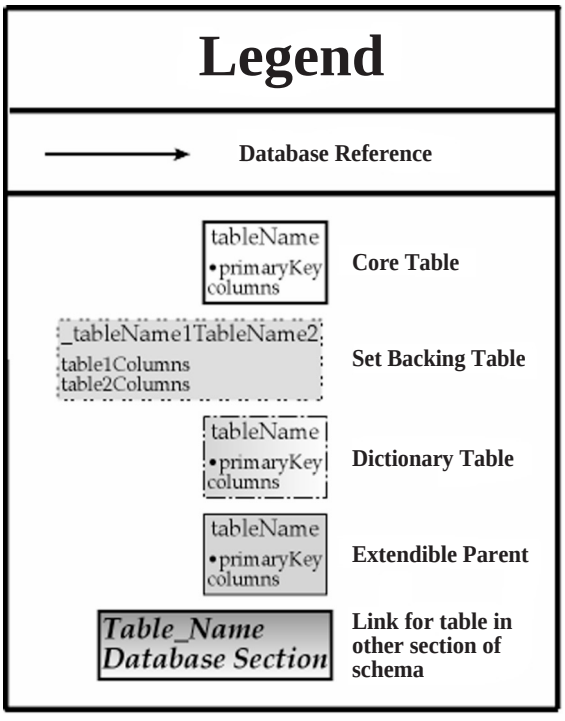


Figure 6 The legend used in describing the database tables and the relationships as represented within BMW.

models may wish to establish a protocol specific to saccadic models. Another modeler may work only with arm-reaching protocols.

As seen within NeuroCore (Chapter 3.2), we have created a table *ids*. This table is used for the generation of unique IDs needed by the database for the storage of entries. In particular, it is used with the *model*, *modelVersion*, *module*, and *moduleVersion* tables to simplify establishing certain relationships. For example, clumps may be linked to the above tables easily by relying upon the *ids* table.

Person

Within the database, there are several different kinds of persons. Consider first a document used as documentation for a model. This document will have one or more authors. If the document is a book chapter, we will also want to store information about the book's editors. Saving the author (or editor) as a person within the database allows us to search the database for other documents this author (or editor) may have written. Another kind of person within BMW is a reviewer, an individual who may be called upon to review models submitted to BMW. We therefore created a table, *person*, within our database (Fig. 7):

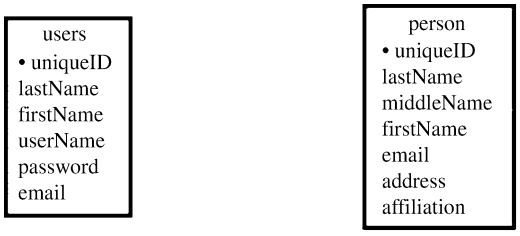


Figure 7 The tables describing people associated with the database. The *person* table describes people stored within the database (such as authors of articles), while the *users* table represents users of the BMW database.

person: A core table that stores information about model and module authors and reviewers. This information includes the person's name, email, current address, and affiliation.

We also need to keep track of the users within the database. Which users have permission to enter data vs. run models? Is the user permitted to create, store, and alter models within the database? Is the user working in conjunction with other users on a model? Because users are treated differently than persons, we created a table, *user*, for describing them:

users: A core table that stores information about the authorized users of the database. These are people who may make changes to the database, as opposed to the general public who may just browse the entries. This information is limited to the user's name, email address, and password.

Models and Modules

A model can be organized in many ways. A model is a packaged, executable program that may be used to run experiments. A module is a logical segment of code that has clearly defined inputs, outputs, and behavior and as such is sufficiently decoupled from the model that it could be re-used in other models. Models are built using modules, which in turn may consist of submodules linked in a hierarchical manner. For example, a model of just the basal ganglia can consist of several modules (Fig. 1 of Chapter 1.1). These modules represent the different regions of the basal ganglia (putamen, globus pallidus, substantia nigra). In turn, the module globus pallidus may be made up of submodules (e.g., its internal, GPi, and external, GPe, segments). This method of organization makes the code not only easier to read, but also easier to plug into other models in development. A motor control model may require the putamen, whereas a saccade model requires the caudate as basal ganglia input, but both models may contain the same representation for substantia nigra pars compacta (SNc) because that brain region is known to project to both putamen and caudate.

Several versions of a model can exist. An early version of a basal ganglia model may only have the direct pathway projection (from caudate to substantia nigra pars reticulata, SNr), but a later version may add the indirect pathway (from caudate to SNr by way of a projection to subthalamic nucleus, STN, and GPe). An important point is that modules and models are versioned independently. This is to allow for independent upgrades or changes to the items without necessarily changing everything. Obviously, a model using a new version of a module must itself be an upgrade to a new model version; the reverse, however, need not be true. A specific model implementation uses a specific version of a given module, but other versions of the model might not use that particular module or use a different version thereof. Also, a module version could be used in any number of models. A module representing the caudate, for example, may be used in several different models of saccadic eye movements. This may occur when a module captures the features that a user desires without the need for modification.

Taken together, tables representing a model include (Fig. 8):

model: This is the highest-level description of a model and is defined as a core table. As such, it contains the most basic information regarding a model: the name of the model (name), a description of the model (what does the model represent?), a reference to information about the owner/creator of the model (ownerID), its creation date (created), and the date when it was last modified (modified). It is to this base table that all other model-related tables must be linked. Because this is the top-level table for describing a model and the modules it contains, the information stored here does not change for subsequent revisions of the model. For example, the Dominey and Arbib model of saccade generation may have multiple versions in which new brain regions are added, but the basic model hypothesis and protocol are unchanged. The only field that changes for this table over time is the modified field.

modelVersion: This table identifies a specific version of a model. A specific version need not be created by the person who originally built the model; we use this table to document who is responsible for the current version via the ownerID field. A model version also has a creation date (created) and last modified date (modified) and a note field for recording information specific to the version. Typically, new versions of models consist of three numbers separated by periods (e.g., 3.2.1). These numbers represent the major, release, and revision information of the model. A version is thus described by the major, release, and revision fields.

modelOSSpecific: Here we store information about a specific version of a model for a specific operating system or implementation. A specific model version may be implemented in various ways (e.g., C code, a simulation program such as NSL, etc.) but accomplishes the same

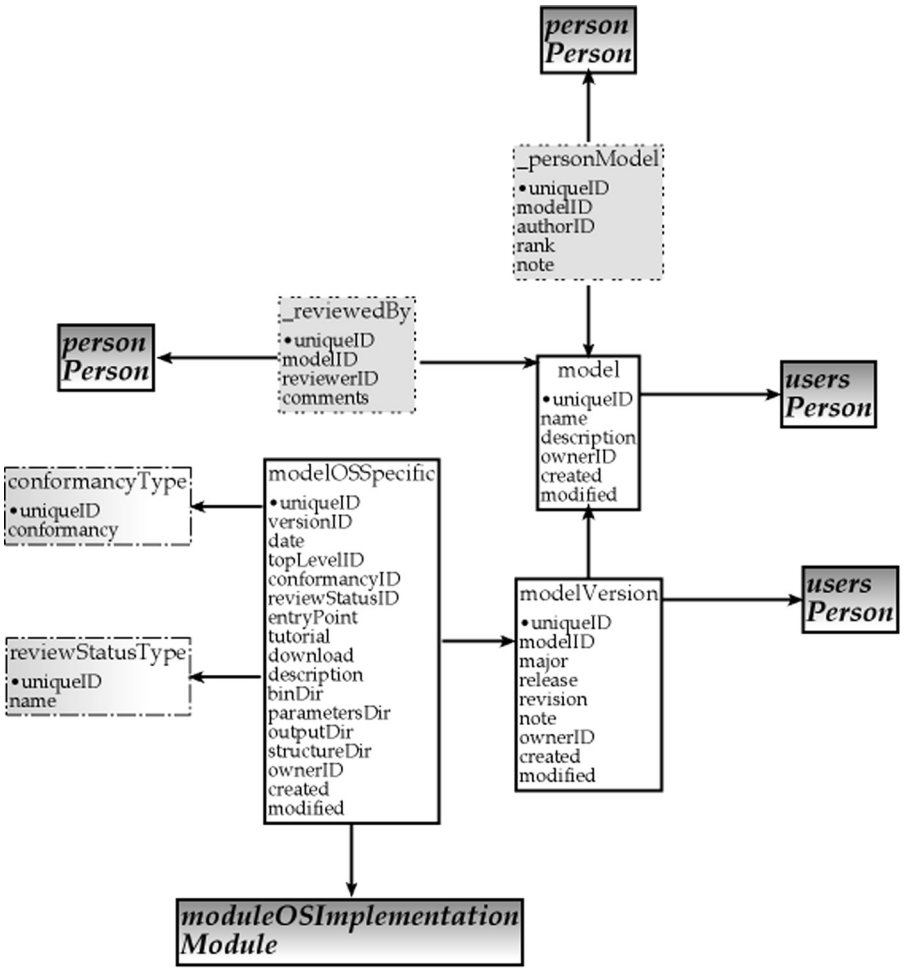


Figure 8 The tables describing a model within BMW. A *model* contains basic information which is expanded upon by the *modelVersion* and *modelOSSpecific* tables. These tables are useful in describing multiple versions of the model, which may vary due to model detail, the implementation (due to operating system or programming language used), and other features.

task and adheres to the same principles and hypotheses. It is not, therefore, a new version of the model but rather a different implementation. Some programming languages are not standardized across operating systems; therefore, alternate versions of the same model code are needed. This table also provides us with the model's version number (versionID), its conformancy (conformancyID) status (is the model a work-in-progress, poster, etc.?), its review (reviewStatusID) status (has the model been reviewed?), and numerous files associated with the model. We also have the owner of this version (ownerID) defined within this level. While a model may initially be created by one person, others are free to modify it and create new versions; we therefore need a method of documenting the person(s) responsible for each version, as well as a description as to how this model is different from other versions. The data files to which this table refers (e.g., tutorial, binDir, parametersDir, etc.) contain documentation and code related to the model and can include the locations of various data files with which the model interacts (e.g., parameters and outputs), a description of the model (which may include information as to why this version is different from another version), and so on.

reviewStatusType: This dictionary table stores all the options for defining the review status of a model. Examples are "not reviewed," "admin review," etc. All work-in-progress and models waiting for review models are therefore of the status "not reviewed;" posters may be either "not reviewed" or "admin review" (meaning they are in the review process); and bronze, silver, and gold models are "reviewed."

_reviewedBy: This is a set backing table used to define the many-to-many relationship between multiple reviewers (reviewerID) and models (modelID). Set backing tables are ways of defining relationships between entries found within separate tables. This set backing table relates models and persons (or reviewers). Thus, a model may be reviewed by many reviewers; correspondingly, a reviewer may review many models. A review may also include comments by reviewers.

conformancyType: This is a dictionary table that stores the options for defining model conformance to BMW requirements. Currently, this includes WIP (work-in-progress), poster, bronze, silver, and gold.

_personModel: This is a set backing table used to encode the many-to-many relationship between authors (authorID) and models (modelID). Thus, a model may have many authors, and an author may own many models through the use of this table.

Fig. 9 illustrates the tables involved in the representation of a module:

module: This is the highest-level description of a module and is represented with a core table. This module consists of a name, a description, its creation date (created), its last modified date, and the module's owner (ownerID). The libraryID field allows us to define a module as belonging to a particular library of routines. Similar to the *model* table, it contains the generic information for describing a module, regardless of the number of versions or alternate implementations of the module.

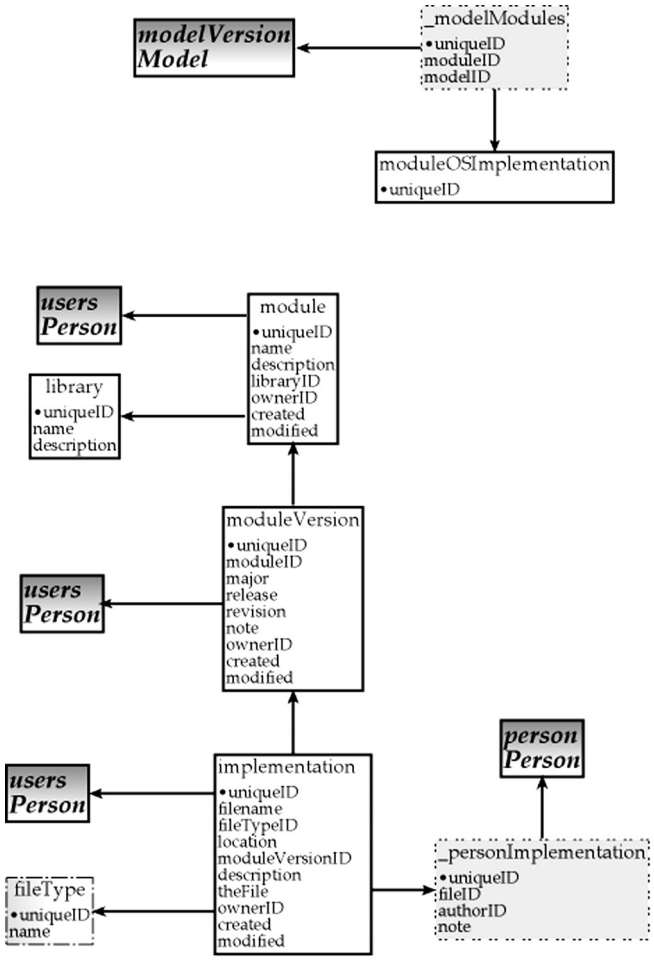


Figure 9 Tables describing a module within the BMW framework. A *module* is the basic building block for a model. The *moduleVersion* and *implementation* tables handle module versions and implementations in a similar fashion to that seen with the model-related tables.

moduleVersion: This core table identifies a specific version of a module and resembles the *modelVersion* table; it contains fields describing the version (major, release, and revision), a note field, the creator/owner of the module (ownerID), the creation date (created), and the modify date (modified).

library: Modules can be grouped in libraries. This core table stores the names and descriptions of a library. A library is represented by a name and a description of what the library represents.

_modelModules. A set backing table that associates models (modelID) and modules (moduleID), this table allows us to define many-to-many relationships between models and modules. One of the key features of BMW is the reusability of modeling code. With this table, we can establish relationships allowing different models to use the same module, and a single model can contain multiple modules.

implementation: This core table stores the file information that forms part of the implementation of a specific module version. The implementation gives us a better idea as to the module's version (moduleVersionID), what kind of file it is (Java, NSL, GENESIS, etc.) using the fileTypeID field, and the location of the module's file (theFile). It also describes the owner/creator of the implementation (ownerID), as different implementations of the same module may be programmed by different people. Finally, it also gives the creation (created) and last modified dates (modified).

_personImplementation: This is a set backing table that identifies the author (authorID) of a file (fileID) that forms part of an implementation. This table allows an implementation to be created (or modified) by several people, as well as allowing a single person to create/modify multiple implementations (of any module for which they have permission).

fileType: This is a dictionary table that identifies file types (C, Java, Makefiles, etc.).

The BMW database has currently been constructed so as to store hyperlinks pointing to a particular model, but the model's code is not physically stored within the database. Obviously, these hyperlinks may point to a model created using any computer language or modeling package. The model tables do have a way of representing non-NSL models, which may not be built in a modular fashion as are those written in NSL. The *modelOSSpecific* and *implementation* tables are constructed with the non-NSL model in mind. Each contains rows that are related to storing executables, parameter files, output files, etc., as well as the file type. With the use of SCS, only NSL models will currently be capable of executing within BMW. In the near future, we will be adding an applet field associated with the particular instance of a model (*modelOSSpecific*). Any model that contains a Java

applet, such as the NSL 3.0 models, will be executable by selecting this field within the BMW Internet interface.

Documents and Clumps

SDB data are stored as "clumps" (which at present—but not in the future—take the form of pieces of text) with a reference to the source document. For each clump, a number of keywords may be stored to facilitate searching. Clumps can be linked to models or modules using a standardized descriptor (e.g., "replicates data," "justifies connection," etc.). This allows two-way searching; for each clump, a list of models and modules that are linked to it can be generated. Similarly, for each model or module, a list of clumps that describe data related to it can be displayed.

Below we briefly describe the tables commonly associated with documents (Fig. 10):

document: This core table stores bibliographic information about clump sources. The current table description is based upon LaTeX format. It is used to store all allowed types of documents (see *documentType*, below) within the same table. It also includes the name of the owner of the document (ownerID), the date the document was created (created), and the last modified date (modified). Normally, a document entry would not need modification unless it contained an error.

documentType: This dictionary table defines the different kinds of documents available within BMW/SDB. These documents include articles, booklets, chapters, conference articles, manuals, theses, technical reports, and unpublished documents.

clump. This core table stores a piece of textual data taken from a reference. It contains the identifier that links it to the originating document (documentID), the text of the clump (theText), the document's URL (if the document is available online), the clump's owner (ownerID, which need not be the same as the document owner but is simply the person who inserted the clump into the database), its creation date (created), and its last modified date (modified). As with the *document* table, modification would occur only if there were errors (such as typographical) which occur within the clump.

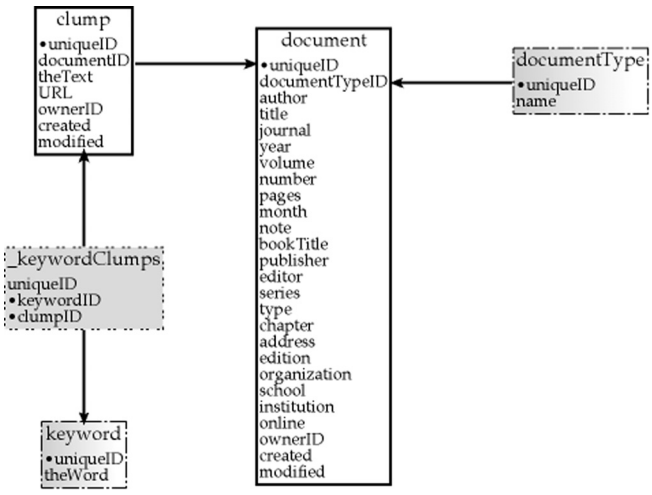


Figure 10 The tables describing documents and their relationship to clumps within BMW. A *document* may be one of several different kinds of documents as defined by *documentType*. A *clump* is a portion of text taken from the document. It may have one or more *keywords* associated with it.

keyword: This dictionary table defines the list of all keywords used with clumps. This allows searches of clumps by keyword rather than by searching the text of each clump. For example, "Patients with lesions to the intermediate and medial cerebellum could not learn to adapt to throwing while wearing wedge prism glasses" would have the keywords cerebellum, learning, and prism adaptation. Users can add as many keywords as they like to a given clump.

_keywordClumps: This set backing table encodes many-to-many relationship between keywords (keywordID) and clumps (clumpID). Thus, a single clump may have multiple keywords, and a single keyword may be associated with many clumps.

Clumps form the basis for data summaries. Presently, clumps are only in text form, but future implementations will involve a wide variety of different data types (e.g., figures, tables, etc.). For models striving to achieve silver or gold status, clumps are essential. They can be used in numerous ways: to characterize the relationships of connections and parameters within a model/module, to document hypotheses, to reference known experimental data which the model is meant to replicate, or to support or refute assumptions made by the model. On the most basic level, a clump contains one or more statements about a topic and links to the reference from which it was paraphrased. We can improve upon this by adding information regarding the relevancy of the clump to the model/module. We have created the following tables to allow us to link with the clump table defined within the SDB (Fig. 11):

_clumpModel: This set backing table associates clumps (clumpID) with either models or modules (atomID). Models/modules may have many clumps of supporting information; conversely, a clump may support several different models/modules.

atomTypes: This dictionary table defines the type of atom, or table type, for tuples in the relevant table. Because *model*, *modelVersion*, *module*, and *moduleVersion* all rely upon a "global" unique ID (generated by the *ids* table), we need a way of determining to what object the clump is attached. This table contains the terms "model," "module," and "moduleversion" so that we know to which item type the clump is related. Without this information, the database would not be able to

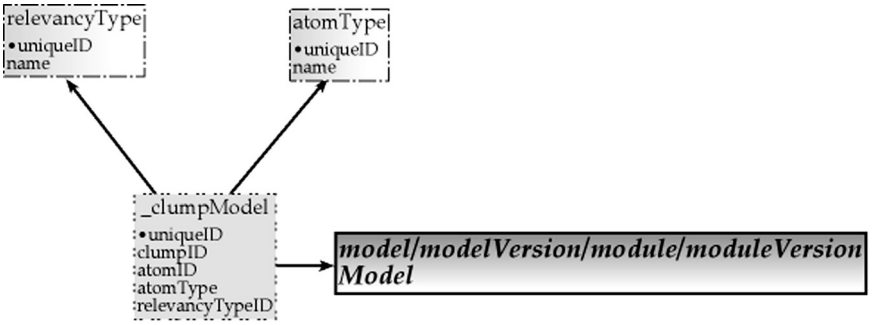


Figure 11 The tables describing the relationships between clumps (found within the SDB database) and objects within BMW. The *atomType* table allows us to define the different object types within BMW (model, module, etc.) that may be linked to a *clump*.

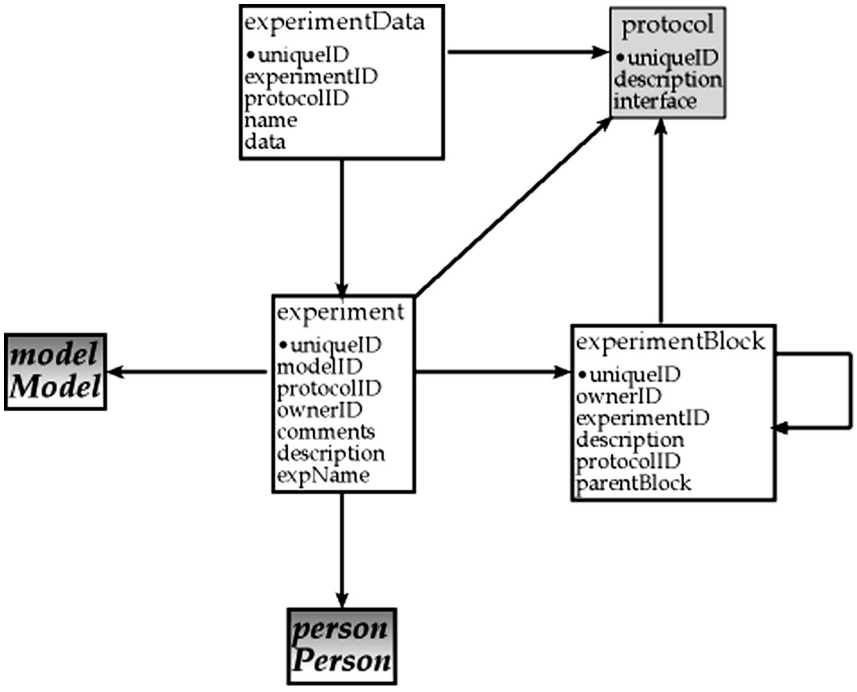


Figure 12 Tables describing the relationship between experiments and models within the database. An *experiment* defines the experiment to be conducted. An *experimentBlock* may define one or more blocks within an experiment (such as seen in learning experiments). A *protocol* may be defined for an experiment, in addition to each *experimentBlock*. Finally, the results of the simulation are stored within *experimentResults*.

follow a clump to a given model/module that uses it for reference (be it support or contradiction).

relevancyType: This dictionary table stores all the strings that are used to describe the relation between a clump and an atom (module/model). Examples include "justifies connection," "replicates results," and "empirically defined parameter."

Experiment

Finally, a model can be used with many experiments. For example, the Dominey and Arbib saccade model can perform simple, memory, and double saccades with only a change to its input parameters. These experiments may include protocols, experiment blocks, and experimental results associated with the experiment (Fig. 12):

experiment: This stores information to execute an experiment using a particular model. Typically, an experiment is defined by the model for which the experiment is conducted (modelID), the creator of the experiment (ownerID), the protocol used for the experiment (protocolID), and the name of the experiment (expName). Sometimes an experiment involves multiple blocks, or

steps, such as is seen in a learning experiment. To accommodate this, we have included a field (expBlock) linking to the table responsible for experiment blocks, *experimentBlock*.

experimentBlock: Experiment blocks may be defined in experiments involving multiple parts. For example, the Dart model involves several blocks: one each for learning to throw overhand and underhand without prisms, blocks for learning to throw overhand or underhand while wearing prisms, and the final blocks during which the model performs overhand or underhand throws without prisms. This table stores the name of the experiment block creator (ownerID), the protocol used (protocolID), and a description of the block (description). Because there are typically several blocks, performed in a sequence, associated with an experiment, the field parentBlock allows us to form a linked list of blocks, with each block also providing a reference to the experiment (experimentID) to which it belongs. Finally, the data associated with each block must be stored. In the Dart model, associated data may refer to the presence or absence of prisms or the target location, among other parameters.

protocol: This links an experiment to a model protocol. Because a protocol may apply to both experiments and experiment blocks, a protocol may be linked with either data type.

experimentResults: This stores results of the conducted experiment/protocol. The links to *experiment* and *protocol* allow us to see which protocol and which experiment generated the results under scrutiny.

6.2.5 Accessing the Database

There are numerous ways to explore BMW and its functionality. We will begin by browsing through the data in order to provide an understanding of how the different tables relate. We will follow this with an example of searching for a particular model and, finally, how to insert information into the database.

Logging into the Database

Upon accessing BMW (http://www-hbp.usc.edu/Projects/bmw.htm), the visitor is first greeted with a brief summary description of the purpose of the database. From this html page, a link is provided to the database. On the left side of the window is a menu used for site navigation. This menu is present for all screens within BMW; at any time the user may click on Home Page and return to this page. There are several items listed within the menu: Model List, for listing the models within BMW; Browse, for browsing the database for models, modules, references, and clumps; Search, for searching the database; Insert, for inserting new information into BMW; Help, a document describing BMW, its tables, and how to use it; and Log In. The screen also provides information to the user about login status and information about the creators and maintainers of BMW. In order to log into BMW, the user clicks the hyperlink titled Log In.

The Log In screen is straightforward. A returning user need only enter a username and a password and then press Enter to access BMW. New users are permitted to request an account using the bottom portion of the screen. The user is prompted for name, email address, and a preferred username. Upon pressing Enter, the user is informed that the request has been sent to USCBP and will be processed shortly. Once users have acquired an account, they can add models to BMW. This account is BMW specific; while the USCBP will have a general account and password, allowing a user to browse all the databases existing within USCBP, the BMW account allows the user further access into BMW. That is, any general user may perform searches or browse the material stored, but only those with a BMW-specific account will be permitted to enter data into the database.

Browsing the Database

Instead of searching for a specific model, module, clump, or reference, users have the option of browsing all the entries in the database. By clicking the hyperlink, Browse, a screen listing the different kinds of data available for browsing is presented. Currently, users can browse models, modules, references, and clumps. We provide a brief description of each in the sections below.

Browsing Models

Fig. 13 is an example list of models stored within the database. This list provides the name of the model, the primary author of each model, and a brief description of the model. Because it is expected that browsing for models will be a common goal of the user, this page may also be reached directly via the Model List hyperlink in the menu. The *model* table (Fig. 8) provides the display with the name, description, and first author of the model. Let's assume that the user is interested in viewing the Dart model. By clicking on the Dart hyperlink, the user is presented with the Web page discussed earlier (Fig. 2) where we see that the Dart model has three different versions: the original version (1.1.1), a version that has added a shoulder model (2.1.1), and a third model with a new interface written in NSL 3.0 (3.2.1) implemented in Java. (Incidentally, this illustrates that BMW is not restricted to storing only models in NSL 3.0.) To display this information, BMW relies not only upon the *model* table for the model name (name) and author (ownerID), but also the *modelVersion* and *modelOSSpecific* tables to describe the different versions of the model (major, release, revision, etc.).

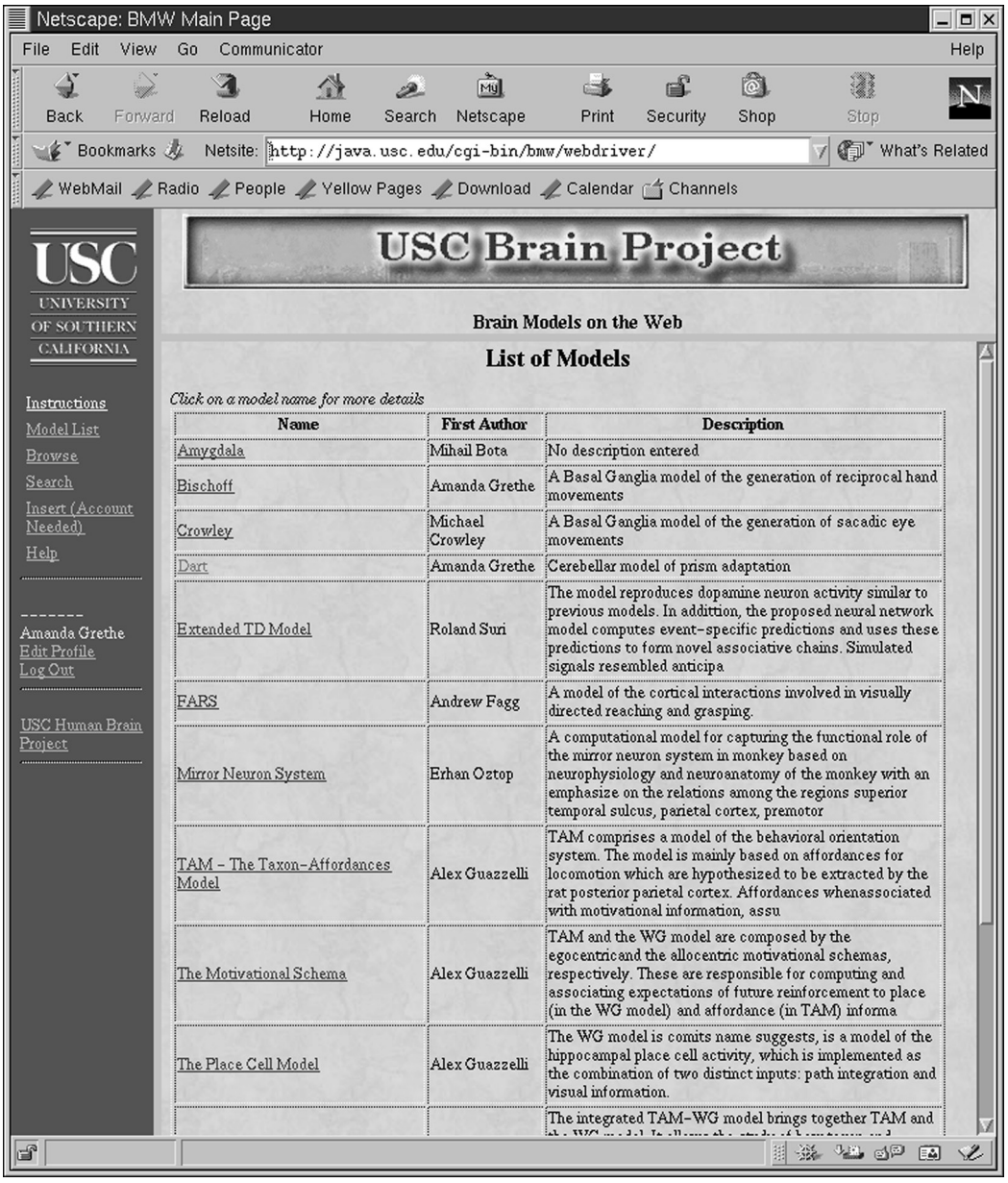


Figure 13 A list of models within BMW. This page may be accessed through either the Browse or Model List links in the menu. The models list relies upon the *model* table to display the name, first author, and description of models stored within BMW. Selecting the hyperlinked model name will take the user to Web pages providing more details about the chosen model.

BROWSING MODULES

To browse existing modules within BMW, the user chooses the Module option from the Browse Page. This will display a new page (Fig. 14) containing a tabular list of all the modules currently stored within BMW. The information displayed includes the name of the module and a brief description, obtained from the *module* table (Fig. 9). For further information regarding a module, the user clicks the hyperlink for that module. For example, clicking the DART_top hyperlink brings up a new page showing a list of versions that exist for this module and a hyperlink for viewing clumps related to the module. The information regarding module versions is stored within the *moduleVersion* table. Using the *_clump-Model* set backing table, we are able to find which clumps (stored in the *clump* table) are associated with the module.

BROWSING REFERENCES

Browsing references is useful should the user wish to see what references already exist in the database without doing a search on keywords or authors. Choosing the All References hyperlink on the Browse page brings up a new page listing all the references currently in BMW. Reference information (found in the *document* table; Fig. 10) displayed includes the name of the author(s)

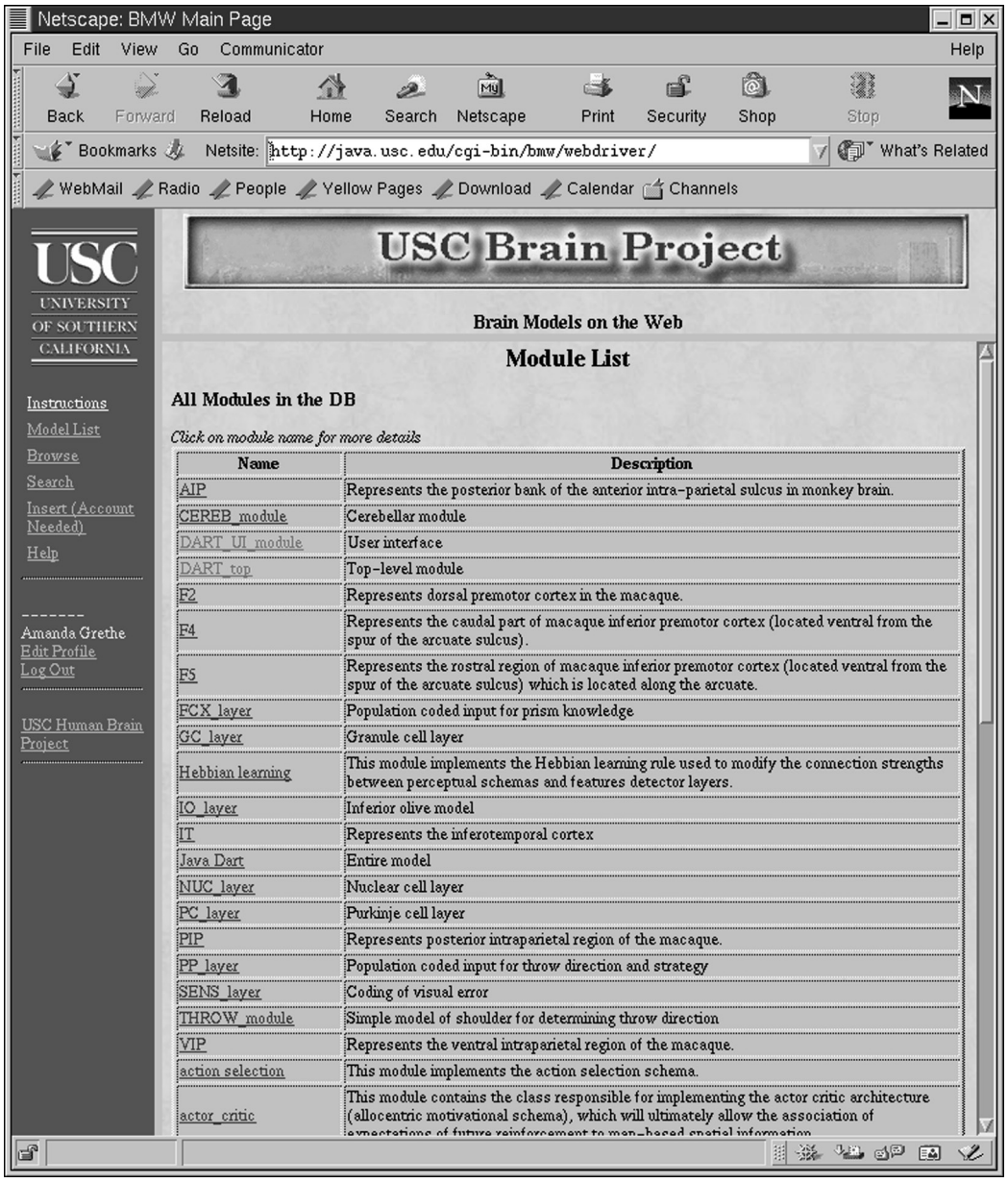


Figure 14 Browsing the modules list. Here, the different modules and their descriptions are presented when the user chooses to browse modules via the Browse hyperlink. The information displayed originates from the *module* table. Selecting a module name will display a new Web page with more details regarding the module.

(author), the year of publication (year), the title (title), the journal (journal), the volume (volume), and the page number (pages) of the reference (Fig. 15). Users can then view a reference more closely by clicking on its hyperlink. The user is told the type of reference (in this case, a journal article) as well as the publication information, provided by the *documentType* and *document* tables. Additionally, the user is also shown a list of clumps related to the article. The *clump* table contains a field titled documentID; it provides the identity of the document from which a clump was obtained. This field is used to list all clumps in the database associated with a particular document. A reference need not have clumps in order to exist within BMW; it merely needs to be entered by an authorized user.

Browsing Clumps

Finally, users can browse clumps within the database by clicking the All Clumps hyperlink on the Browse page. This brings up a page listing all the clumps that reside within BMW, as described by the *clump* table (Fig. 10). The user is given the name of the source document (via the *document* table) and a title for the clump and is also given the opportunity to view the clump in closer detail by clicking on its title hyperlink. Details of a clump (Fig. 16) include its source reference (documentID), the

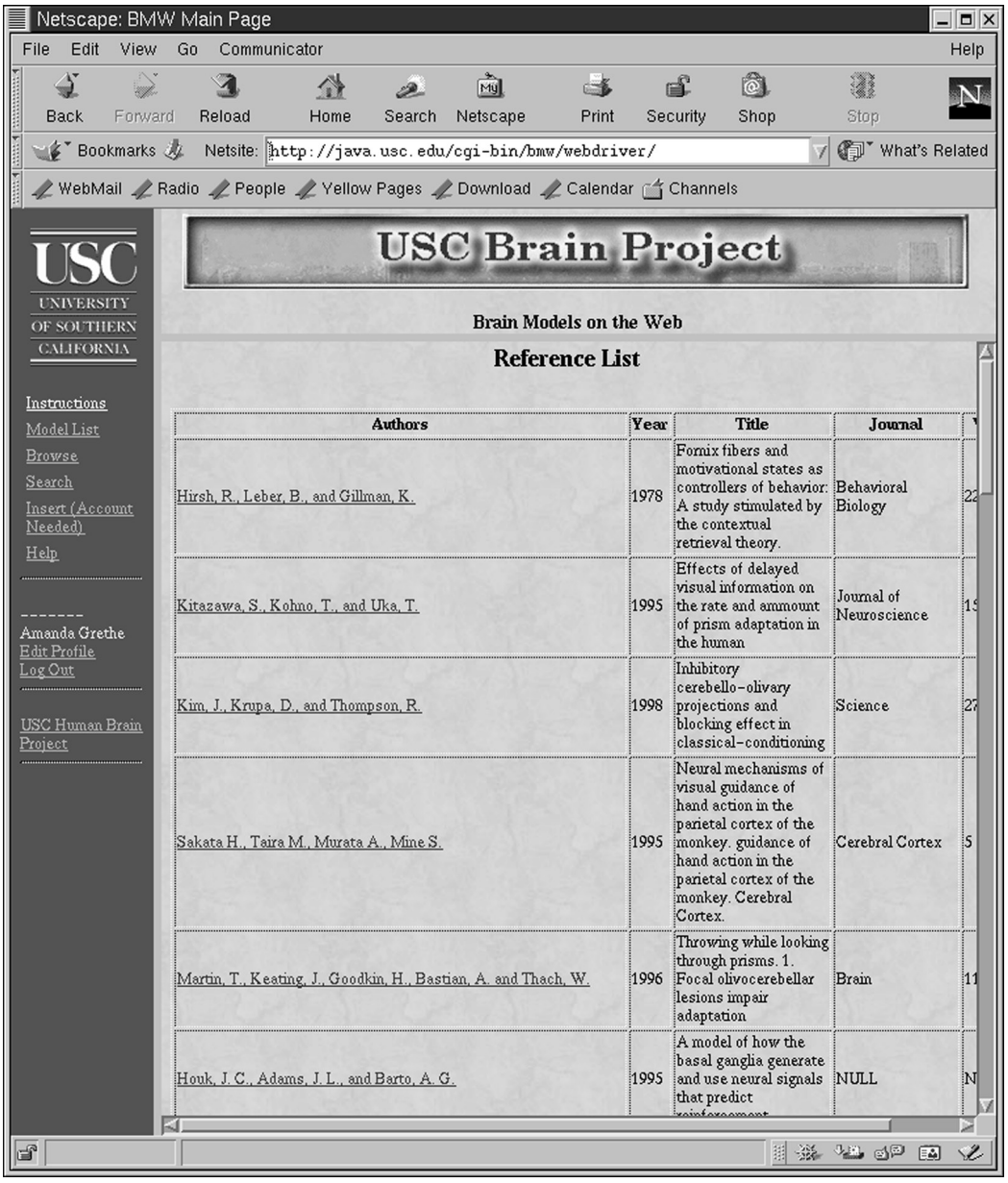


Figure 15 Browsing the reference list. The user is presented with the following reference information (found in the *document* table): the authors, year, title, journal (or other document as appropriate), volume, and the page number of the reference. Selecting the author hyperlink will create a new Web page with further information about the selected document, such as clumps related to the document.

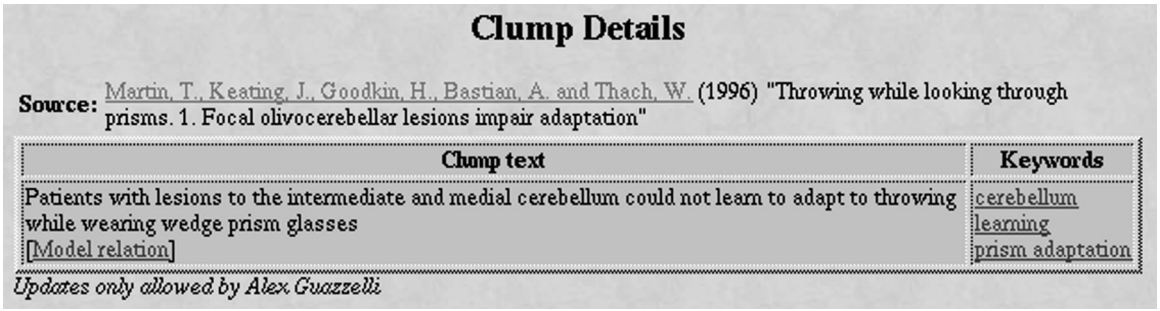


Figure 16 Details of a clump. The user is presented with the author(s), year, and title of the article from which the clump was taken. The compete text of the clump is also presented, along with the keywords associated with the clump and a hyperlink to the list of models and modules that use it.

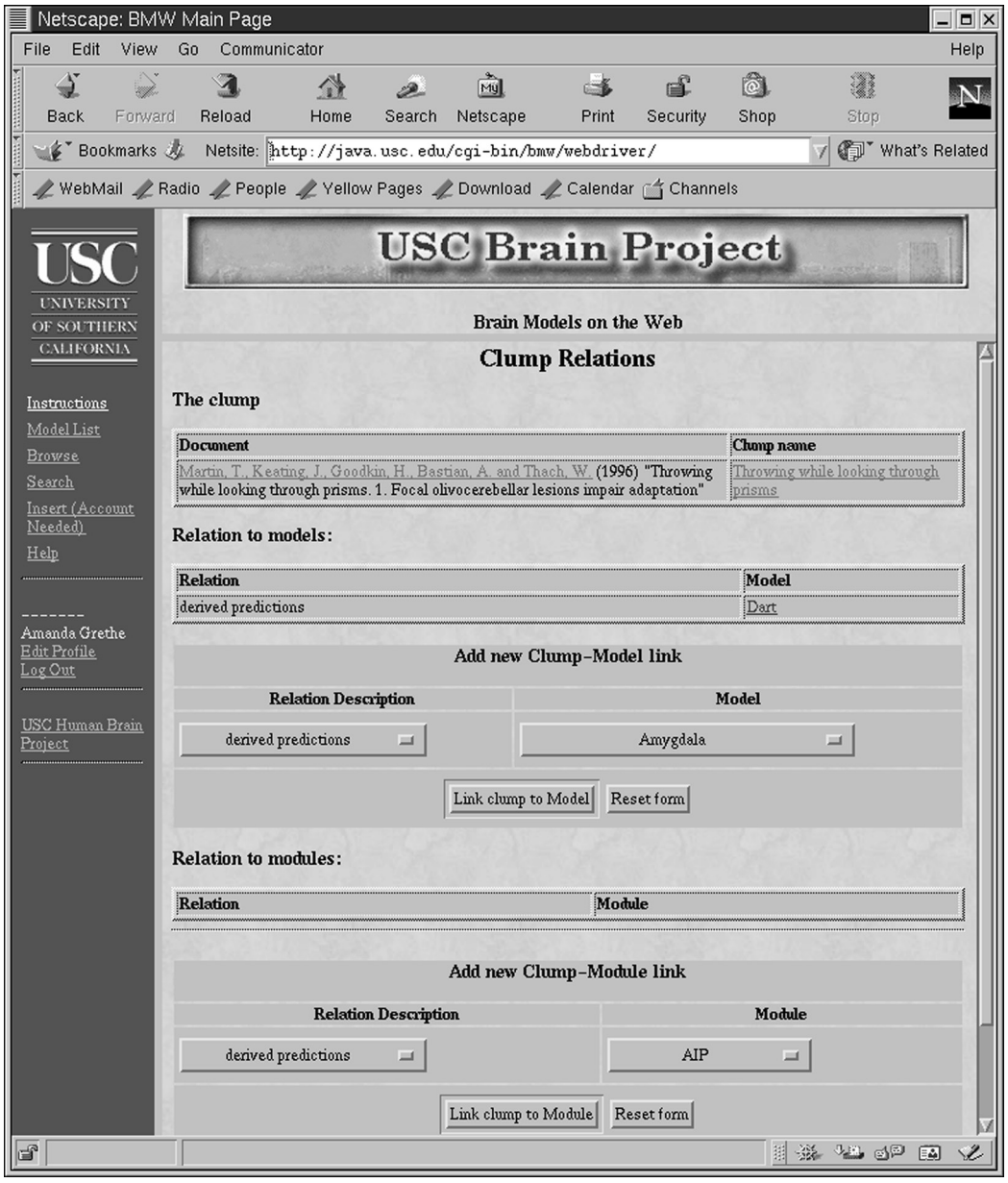


Figure 17 This clump can also be used to illustrate the precise relationship between the text and the model(s) that used it. Here, the Dart model is shown to derive predictions from the selected clump of text.

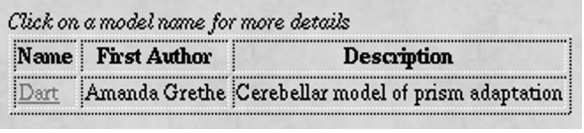
complete text of the clump (theText), keywords associated with the clump (via the *_keywordClumps* set backing table), and a link to the models and modules linked to the clump. The latter link exists both on the Clump List and the Clump Details pages. Clicking on this link displays a screen describing which models are associated with the chosen clump (Fig. 17). This is determined by the *_clumpModel* set backing table; given a uniqueID of a clump, it provides a list of identifiers for models which use the specified clump. In the example shown, the clump related to prism throwing is associated to the Dart model and was used to derive predictions. The pull-down menus below allow the user to add links between clumps and models and between clumps and modules.

Searching for Models and Data

Users are able to search the database for three different kinds of data: models, clumps, and documents. Upon clicking the Search link within the menu, the user is presented with the entry page for search capabilities. The different searches are presented below.

Searching for Models

To search for models from the Search Page, the user chooses the Models hyperlink. This brings up a search screen related to models. The user is given the option of searching for either a text string within a model's title or for a specific model author. Depending upon the search chosen, different fields will be checked for the search


Click on a model name for more details

Name	First Author	Description
Dart	Amanda Grethe	Cerebellar model of prism adaptation

Figure 18 The result of a search for a model containing the word "Dart" in its title. In this example, only one model satisfies the search criteria.

term; a text string search will check the name field of the *model* table, while an author search will rely upon the ownerID field of the *model* table (Fig. 8). In our example, the user has chosen to search for all models containing the word "Dart" in the title. The search is initiated when the user clicks the Word Search button, and the next screen displays the search results (Fig. 18). In this case, there is only one model containing "Dart" in the title.

Searching for Clumps

A user is able to do searches for clumps in three ways: search for a text string contained within a clump, search by a keyword substring, or search by choosing the keyword from a pull-down list of keywords. A text string search within a clump will involve searching the Text field within the *clump* table. This can be very time consuming, but it is useful if the user is looking for something that may not exist as a keyword. Keyword searches are quicker, whether using a predefined list or a user-specified substring. Keywords available within the database are stored within the *keyword* table (Fig. 10). The *_keywordClumps* set backing table is searched for the entered keyword (keywordID) and links to the *clump* table (via clumpID) to return a list of clumps associated with the given keyword. For example, if the user chose to search by the keyword "prism adaptation" and then clicked the Find button, the *_keywordClumps* table would be used to determine a list of clumps related to prism adaptation. Upon submitting this search, the user is presented with a new screen listing clumps that contain it. This list includes the source document, a portion of the clump of text, and all keywords associated with that clump. The user can continue searching on different keywords by clicking on the desired hyperlinks on the right-hand side of the screen. Clicking on the clump text will bring the user to the screen describing the clump as previously seen in Fig. 16.

Searching for Documents

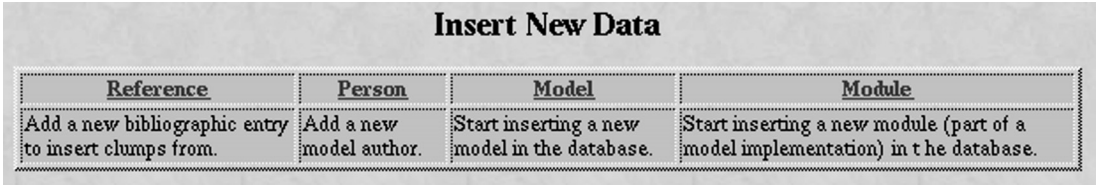
Finally, users can search for specific documents within BMW. By choosing the Documents hyperlink on the Search page, the user is presented with a search screen related to references. Users can currently perform a search by author or by a text substring. For example, a search for all articles within BMW that are authored by "Thach" will use the author field of the *document* table to bring up a new page listing the articles co-authored by Thach that the database currently contains. Clicking on the hyperlink for a given reference brings up a new display giving document details and listing related clumps. Searching by a text substring will involve searching the title field within the document table.

Inserting Models and Data

Before data can be entered into the database, the user must first log in (see above for details on the login process). To begin inserting data of any kind within the database, the user selects the Insert hyperlink from the menu. A page will appear that details the different kinds of data that may be entered into BMW (Fig. 19). Currently, users may choose to insert references, persons, models, and modules.

Inserting References

From the Insert New Data page, the user selects the Reference hyperlink. The database allows the entry of a large number of classes of bibliographic references, modeled after the format used by LaTeX. The supported reference types include articles, booklets, chapters, conference articles, manuals, theses, technical reports, and unpublished documents (as defined within the *documentType* table; see Fig. 10). In the current example, the user has chosen to enter a journal article. An entry screen appropriate for this reference type appears (Fig. 20). The On-line field is for an optional URL to an online version of the reference. Entering it without the "http://" prefix (e.g., www.name.org/here/this.html) would be a valid entry. The rest of the fields should be self-explanatory; they include the author, title, year, journal name, volume, number, and pages for the article, along with any notes the user may wish to enter regarding that


Insert New Data

Reference	Person	Model	Module
Add a new bibliographic entry to insert clumps from.	Add a new model author.	Start inserting a new model in the database.	Start inserting a new module (part of a model implementation) in t he database.

Figure 19 The start screen, which enables users to insert new data of various kinds. Currently, insertion of reference, person, model, or module is supported. In order to insert one of these data types, the user must be logged in and must select the appropriately titled hyperlink.

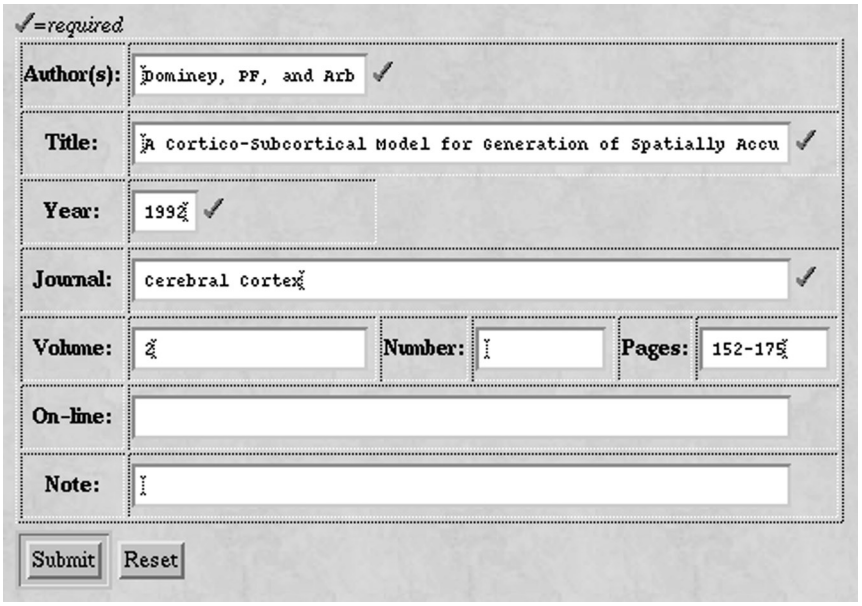


Figure 20 The entry page for storing a new article reference within BMW. At a minimum, the user is required to enter the author(s), the title, the year of publication, and the journal name for insertion within the *document* table.

Figure 21 An example of a model being entered into BMW. The minimum requirements are a name for the model and the name of the first author. Model authors are available from a pull-down list of persons stored within the database. The data are then stored within the *model* table.

article. Once the user presses the Submit button, a message appears telling the user if the insert was successful or if there is information missing from the entry. The entered document is added to the *document* table within BMW.

Inserting Persons

Although a model is referenced with its first, principal, author, there may be multiple authors involved with that project. We have therefore provided a method for adding new people to the database who can later be attached to the model. Note that the person is not the same as the BMW users list. From the Insert New Data page, the user chooses Person. This will bring up a page for entering a new person. The minimum required information about a new person includes the first and last name and email address. The user can additionally add the address and the affiliation of the new person. Entered information will be stored within the *person* table.

Inserting Models

From the Insert New Data page, the Model hyperlink leads to the pages used for inserting a model and its modules into BMW. The minimum requirements for the *model* table are a name and the first author (Fig. 21). Note that the Author entry is obtained by selecting from a menu, based upon the names stored within the person table. If the principal author's name does not appear in the menu, it has to be inserted into the database first (see "Inserting Persons"). When the information is entered, the user presses the submit button. The model name will now be entered into the database.

Once a model exists, the user needs to establish the initial version of the model. After a model is entered, the user is presented with a page that allows entry of versions (Fig. 22). After pressing the hyperlink New, the user can begin entering a new model version. When presented with an Insert Version screen (Fig. 23), the user may enter a version number and add notes detailing how the

Figure 22 The screen for inserting new versions of a model. The model name, description, and author (found in the *model* table) are displayed. Previously defined versions of the model may be listed in the table. To insert a new version, the user selects the New hyperlink.

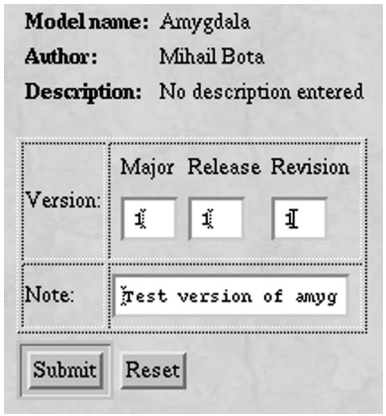


Figure 23 An example of entering a new version for a model. A version is defined by three fields: major, release, and revision. The user can change the version number and add notes describing any difference between this and previous versions of the model.

model is different from previous versions (should they exist). This is stored within the *modelVersion* table, including the modelID that was created when the model was inserted into BMW. Once the version information has been submitted, the user can begin entering model implementations.

The implementation of a model can refer not only to the modules comprising a model, but also to other aspects, such as the machine, the operating system, the compiler, and the graphics environment. We can add new implementations of a specific model version as the need arises. After clicking the New hyperlink while looking at a model version, the user is then presented a page for entering implementation information (Fig. 24) based upon the fields within the *modelOSSpecific* table. The user is given the option of entering a description, the review status, its conformancy (defined in the *conformancyType* table), and the locations of the executable and tutorials on the model. In order to add the executable code and the tutorial, a URL must be provided. BMW will store the URL, allowing later users to run the model or read the tutorial. Note that the executable and the tutorials are not explicitly stored within BMW.

After the model implementation has been established, the user can now enter modules. From a model's

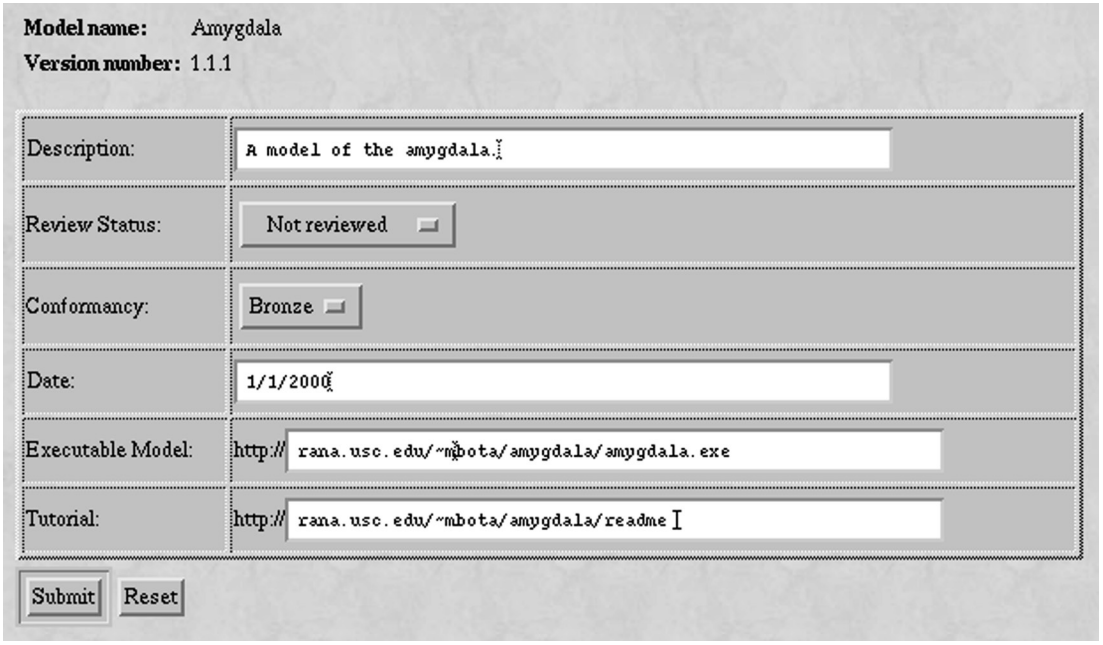


Figure 24 The user is invited to enter pertinent information regarding the implementation of the model. This can include the locations of the executable version of the model and a tutorial explaining how to use the model. This information is stored within the *modelOSSpecific* table.

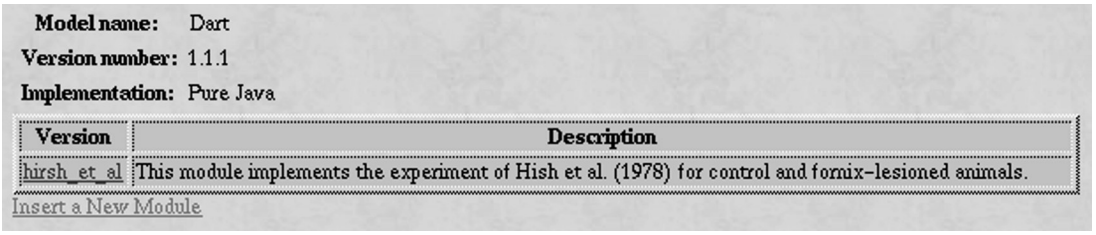


Figure 25 The module list screen. Modules previously defined for a model will be listed within the table. Clicking the Insert a New Module hyperlink allows the user to create new modules associated with the model.

Figure 26 The new module entry screen. The user is required to name a newly defined module and is able to enter a brief description. The entered information is stored within the *module* table.

implementation (Fig. 25), the user may click on the Insert a New Module hyperlink to bring up a screen for entering modules (Fig. 26). The module insertion screens follow a logic similar to that seen for inserting a model: create the module (using the *module* table); create a version of the module (using the *moduleVersion* table); and, finally, define the module's implementation (using the *implementation* table).

Inserting Clumps

Clumps are short summaries of information contained in a reference. Before one can enter a clump, the source document must exist as a reference in the database. New Clumps are entered using the form on the page displaying details of a reference; therefore, the user must first find the document and then click on the author of the clump. All documents are listed within the *document* table, and those people capable of entering clumps are listed within the *person* table. Clump text is entered in the text area (Fig. 27). A URL may be entered for referral to a page with more information on this server in the user's personal space. The default path is set to "~bmw/contrib/users/[user login]/" in case users would prefer to upload the pages to their user spaces on our server. In this case, users must use ftp to post the pages at the specified locations.

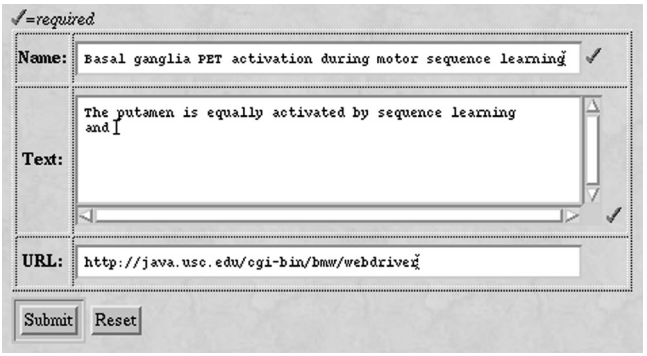


Figure 27 The clump entry screen. The clump is given a name, and the text is quoted within the text area provided. The user may also enter the URL at which the originating document may be found. The data are stored within the *clump* table.

6.2.6 Future Plans

Links to SCS

Most of the features of BMW-SDB can be exploited for any model that has a modular structure. BMW currently includes models written in modular fashion using Java as well as NSL. However, the current version of SCS is written specifically for the development and execution of models written using NSL 3.0 (Chapter 2.2) and thus provides features for BMW-SDB that are not available for models developed with other simulators. SCS is therefore useful if the BMW user wishes to run or modify a NSL model. To execute a NSL model, the user must first download the SCS/NSL software onto her client machine and start SCS. When SCS downloads a model, it will be saved as a set of module files within the user's account to reduce communication and database access time. A NSL module typically contains one file with the extensions .nsl (NSL scripting language), .mod (NSL modules), and .sif (schematic and icon file). If it is intended for execution on the NSL 3.0 system implemented in Java, then it will also include a .class file, as well. If

it is intended for execution on the NSL 3.0 system implemented in C ++, it will also include a .o and .exe file. The current implementation of SCS provides access to a library of models and modules and tools for model development. Work currently under way will extend SCS so that, should the model also contain summaries (i.e., the model is silver or gold), documentation from the SDB will be viewable via the database browser within as well as (the case documented in this chapter) outside of SCS/NSL.

After making modifications to an existing model, the user may also choose to upload a model to BMW via SCS. If the model has been changed, it would need to go though the review process again. To make major updates known, the user must choose to submit these updates to the poster session. When the SCS/NSL session ends, the user will be asked if the current work should be returned to BMW as a new version. If the user elects to do so, the files will be stored in the appropriate manner. SCS/NSL will also have the capability to submit documentation to the database; alternatively, users may work with BMW using their favorite Internet browsers. In the examples throughout this chapter, we have worked with BMW via the Web interface.

Links to Experimental Neuroscience Data

The NeuroCore schema (Chapter 3.2) describes a way to organize large datasets from the neuroscience community. These diverse datasets provide the details needed by a computational model if it is to capture the behavioral phenomenon in question. Currently, BMW relies upon summary statements that describe the experimental data; it also relies upon published results for a graphical comparison of the modeled and experimental data. One significant method of extending BMW is to allow it to link to the experimental data stored within NeuroCore. This would serve several purposes: (1) it would allow for a direct comparison between experimental and modeling data; (2) it would directly provide the data necessary to determine the model design; (3) model results for particular simulations could be stored within the NeuroCore framework; and (4) it would further serve as model documentation (e.g., a simulated neuron matches parameters from a neuron studied *in vivo* and stored within NeuroCore). This enhanced functionality would lead to a more powerful tool in the development and testing of computational models and would serve to better inform the neuroscience community on the linkages between simulated and experimental datasets. The tables in BMW that represent experiments and results have yet to be implemented. Once complete, these tables will be used in comparisons between NeuroCore data and simulated data.

Links to Synthetic PET

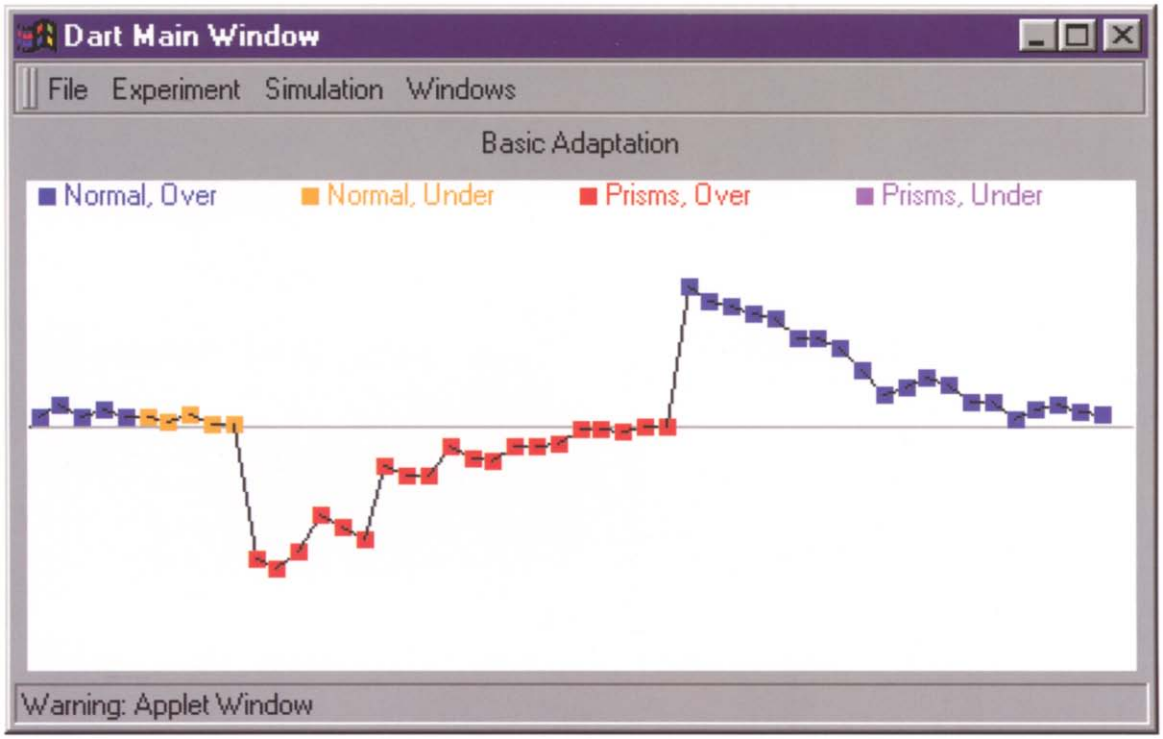
Synthetic PET, as described in Chapter 2.4, is a method of calculating PET images from a computational model. These images may later be compared with human PET data for later model refinement or can assist in the development of new experimental ideas. Synthetic PET relies upon the overall synaptic activity generated by the model's connections. At this time there is no direct interface between BMW and Synthetic PET; rather, Synthetic PET is a standalone applet that accesses the BMW database for its synaptic data. We would like BMW and Synthetic PET to be more fully integrated, such that a model being viewed within BMW can automatically launch a Synthetic PET comparison. With plans to include neuroimaging data within NeuroCore, users would be able to use Synthetic PET as yet another way to compare simulated and experimental data.

References

Dominey, P. F., and Arbib, M. A. (1992). A cortico-subcortical model for generation of spatially accurate sequential saccades. *Cerebral Cortex* **2**, 153–175.

Martin, T. A., Keating, J. G., Goodkin, H. P., Bastian, A. J., and Thach, W. T. (1996a). Throwing while looking through prisms. 1. Focal olivocerebellar lesions impair adaptation. *Brain.* **4**, 1183–1198.

Martin, T. A., Keating, J. G., Goodkin, H. P., Bastian, A. J., and Thach, W. T. (1996b). Throwing while looking through prisms. 2. Specificity and storage of multiple gaze-throw calibrations. *Brain* **4**, 1199–1211.



CHAPTER 6.2, FIGURE 5 Example of output from the *Dart* model. The graph demonstrates the effects of wedge prisms on underhand and overhand throwing, along with the adaptation of the user's ability to hit the target when a dart is thrown. In this simulation, the model initially throws both overhand and underhand under normal conditions and is fairly accurate. When prisms are donned, the model must adapt to the related change between sight and movement. Once the model has reached an appropriate level of accuracy, the prisms are removed. The model makes errors in the opposite direction before readapting to normal conditions.